481.1001CON2

REMARKS

In response to the Office Action, the title has been amended to conform to the preamble of the claims. A terminal disclaimer is submitted herewith to overcome the overcome the obviousness type double patenting rejection. Another copy of the World Tool Kit Reference Manual is enclosed, in response to the Examiner's request.

Reconsideration and allowance of the present application is respectfully requested.

Respectfully submitted,

DAVIDSON, DAVIDSON & KAPPEL, LLC

By____________________________
Cary S. Kappel
Reg. No. 36,561

Davidson, Davidson & Kappel, LLC
485 Seventh Avenue, 14th Floor
New York, New York 10018
(212) 736-1940

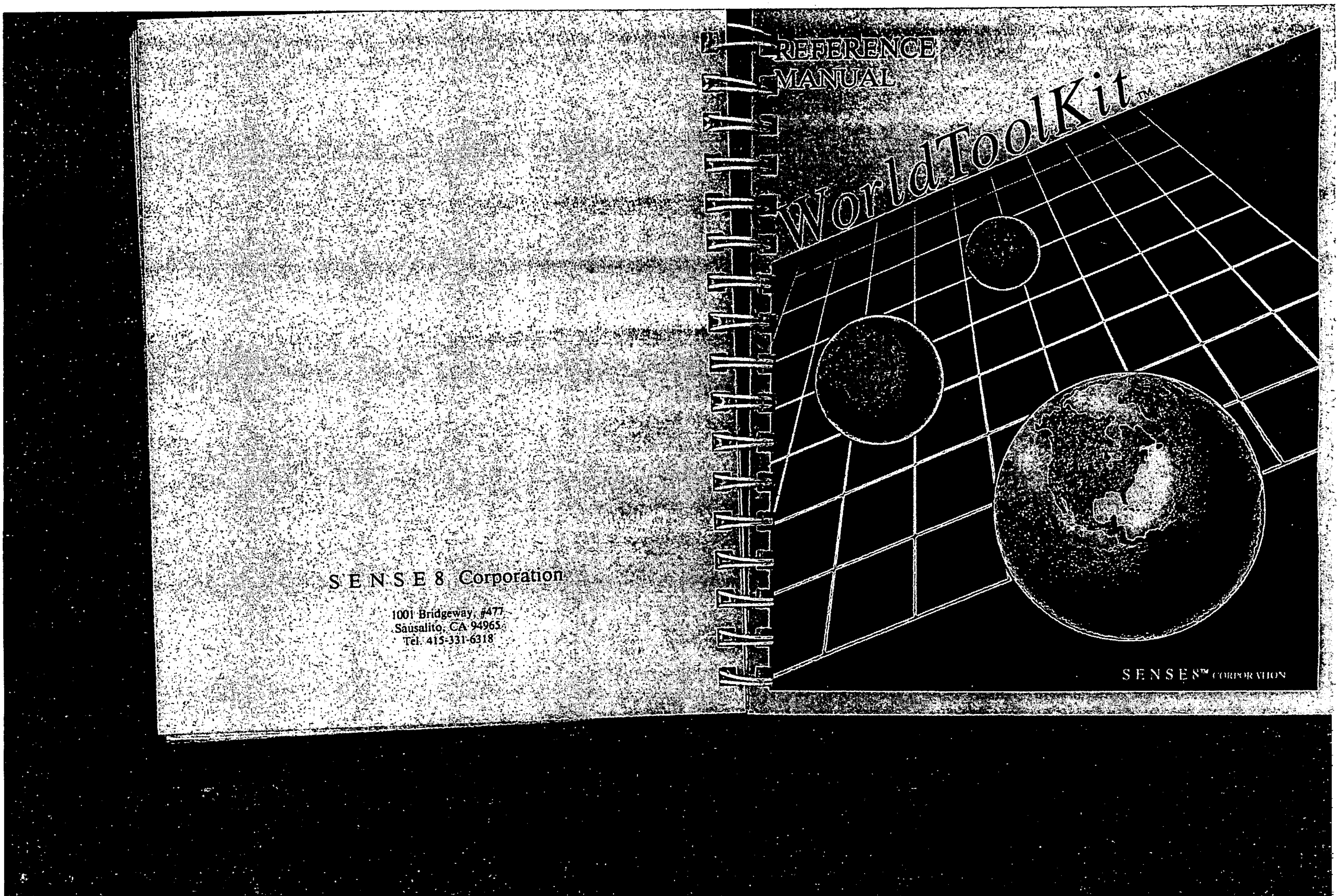

S E N S E 8 Corporation
1001 Bridgeway, #477
Sausalito, CA 94965
Tel. 415-331-6318
REFERENCE
MANUAL
WorldToolKit™
S E N S E 8™ CORPORATION

Table of Contents

Index

Functions

WTK has many different functions that are activated by pressing the keys on the keyboard and responding to the prompts given to you. A useful thing to do first is to press the "spacebar". This brings up the first of two keypress summary screens listing the available functions and their associated keypresses. Press any key to bring up the second screen.

After you have reviewed the two keypress summary screens, you may activate a function by pressing its key. For example, pressing "f" flips WTK between "mouse-move-viewpoint" and "mouse-pick-object/polygon" mode. This keypress is a frequently used one.

Please see the README file in the demo directory for information on how to use WTK.

Lexical Conventions

Throughout this manual, segments of C code, including function prototypes, code samples, and the C variables described in the body of this text, appear in 10 point Helvetica type. For example:

```
universe_new()
```

The body text of this manual appears in 11 point Palatino type.

Mathematical Types and Conventions

WorldToolKit uses a variety of typedefed structs to represent mathematical quantities such as 3D positions (Posn3d) and

orientations (Quat, which stands for quaternion). Such typedefed quantities appear as arguments to many functions. Consult Chapter 11 as needed for clarification of these types and for a complete description of the conventions and calls in the WorldToolKit math library.

Other Documents

See also:

WorldToolKit Installation Guide (Sense8)

ActionMedia 750 Board Installation Guide (Intel Order Number: 505840-001)

ActionMedia 750 Board Owner's Guide (Intel Order Number: 506176-002)

ActionMedia 750 Production Tool Reference (Intel Order Number: 463629-003)

shelf.bak) then WTK will create dynamic objects from table.dxf, chair.dxf, book.dxf and shelf.bak, and will ignore test.c assuming that it does not contain comprehensible geometry. Please review the descriptions of stationary and moving objects in Chapters 1 and 2 of this Manual.

A word of caution here. WTK does some error checking with the command line arguments, but not a great deal. For example, if you include the wrong *-xn* argument to tell WTK where your sensor is attached, then WTK may quietly sit there and not return control back to you. You will need to reboot the computer. Remember, WTK is a simple program to get you started with WTK and not a robust application.

How to build your world

The sample program WTK uses WorldToolKit's utility for understanding many different 3D geometry files, and displaying their contents in real-time. For this example we assume that you are using a modelling program that can generate *.DXF (Autodesk's "Drawing eXchange Format") files from your CAD models.

When we build simple worlds, we typically construct the stationary background object and dynamic objects in the same CAD model, and then write out separate *.DXF files for each one. For example, let's way that we have a simple model of a room with four walls, a floor and a ceiling. In this room we have modelled five objects: a table, chair, book, computer and bookshelf.

If we created a *.DXF file (in ASCII text format) of this model complete with the five objects and called it my_room, then we could start WTK with the following command line:

```
WTK -xn my_room.dxf
```

where my_room.dxf is in the current directory. We would be launched into our virtual world with everything in the room treated as one stationary object (including the table, chair, book, computer and

bookshelf). This form of WTK is really a static "walk-through" program, where you can interactively texture the surfaces of any object, but you cannot interact with the objects themselves.

Now, let's reorganize our files into two directories models and parts, as shown in the following example:

```
c:> dir models
MY_ROOM     DXF   47471 06-02-91  1:20p
1 File(s)   8937472 bytes free

c:> dir parts
TABLE       DXF    3233 06-02-91  1:20p
CHAIR       DXF    9403 06-02-91  1:21p
BOOK        DXF    2333 06-02-91 23:23p
COMPUTER    DXF    3222 06-02-91 23:24p
SHELF       DXF    3231 06-02-91  6:55a
5 File(s)   8937472 bytes free
```

If we now run WTK with the command line of:

```
WTK -xn models\my_room.dxf parts
```

then WTK will load my_room as a stationary object, and the five objects in the parts directory as dynamic objects. You can now manipulate the five dynamic objects.

A note here: When WTK reads a file (say a *.DXF file) and creates a 3D object from it, the object is placed in the virtual world at the position it was saved when you created the original file. In addition, AutoCAD™ saves the current viewpoint along with the geometric information in your DXF file. The source code WTK.C shows you how to use this information to place the viewpoint in your virtual world.

For convenience, you can construct your stationary and dynamic objects in one model (where the dynamic objects are positioned correctly within the model space) and then save each one to a separate file.

In WorldToolKit there can be many viewpoints. The viewpoint created by universe_new is automatically used as the current viewpoint (the one through which the virtual world is displayed).

universe_delete

```
void universe_delete():
```

universe_delete should be the last WorldToolKit call in your main program. universe_delete frees all of the objects in the universe, including those that have been removed from the simulation with the remove function appropriate for that object type, such as object_remove. The universe_delete call also cleans up and closes the graphics hardware.

Call to universe_delete will cause execution of any exit action function established by prior call to universe_set_exitfn.

Universe loading and saving

Two special functions universe_load and universe_save are provided to help maximize the graphics rendering speed and therefore the frame-rate of your application. These functions are used to handle a special type of *stationary* graphical object supported in WorldToolKit. Because this object is known to be stationary, WorldToolKit can intelligently partition the object's geometry into a form that is most efficient for rendering. Examples of stationary 3D geometry might be an architectural model or background scenery. Note that only one stationary object may be loaded in at any one time. -

universe_load

```
FLAG universe_load(filename, modelview, scale)
char *filename;
Posn6d *modelview;
float scale;
```

Use universe_load to load from a file any stationary portions of the 3D geometry which make up your virtual world. The first argument to universe_load is the name of the file you wish to load in. This file may be in DXF format, in WorldToolKit's neutral file format, or it may be a file generated by the Amiga-based modelling programs VideoScape or Caligari. It may also be a binary file created with the WorldToolKit function universe_save. Many other file formats can be loaded into WorldToolKit through use of third party geometry conversion programs capable of writing formats which WorldToolKit can read.

The DOS environment variable WTMODELS gives a path to geometry files which is searched before the current directory for the filename passed in to universe_load. For example, let's say that from the DOS command line you:

```
set WTMODELS=c:\dxf;c:\demo
```

Then, if universe_load is called with filename "oplan.dxf", this file would first be searched for in `c:\dxf`. If not found there, it would be searched for in `c:\demo`. Finally, if still not found, it would be searched for in the current directory.

For each virtual world that you wish to create, universe_load may be called only once. Therefore your entire stationary backdrop must be included in one file. Calling universe_load automatically calls universe_vacuum to make way for the new graphics being loaded in. Therefore, any calls to functions which create dynamic graphical objects, terrain, animation sequences, or portals must occur after universe_load is called.

1 The Universe

Introduction to the universe class

The universe is the container of all WorldToolKit objects. These objects may include graphical objects, sensors, lights, animation sequences, portals, viewpoints, and serial ports. Once these objects are created, they are automatically maintained by the WorldToolKit simulation manager.

Unlike the methods for other WorldToolKit objects, universe methods do not take a handle to the universe as a first argument. This is because there is only one universe in existence at any time.

Universe construction and destruction

universe_new

```
void universe_new();
```

universe_new should be the first WorldToolKit call in your main program. This function initializes the universe's state, initializes the graphics device, and creates a viewpoint for the universe. universe_new must be called before any calls are made to constructor functions for WorldToolKit objects such as sensor_new, object_new, and light_new.

universe_go1 to go through the simulation loop exactly once and then exit the loop automatically. Figure 1.1 shows the order of events in the simulation loop.

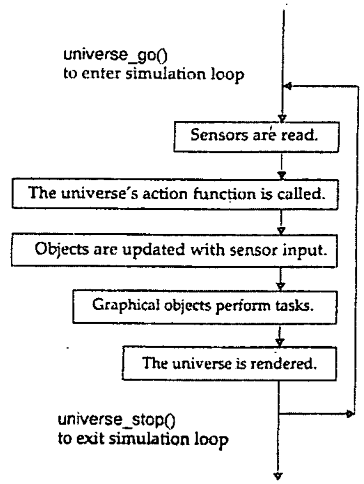


Figure 1.1 The simulation loop.

universe_ready

```
void universe_ready();
```

universe_ready prepares your application for entry into the main simulation loop. It computes overall geometrical properties (midpoint, extents, and radius) of the graphical entities in the universe. These properties are illustrated in Figure 1.2. In addition, universe_ready optimizes the shading model and finalizes colortable settings. Since these computations can sometimes be time consuming, universe_ready should only be called outside of the real-time loop.

universe_ready must be called before starting the simulation for the first time (i.e. before the first call to either universe_go or universe_go1), but after all graphical entities have been created (i.e. after all calls to object_new, animation_new, universe_load, or the terrain functions). Subsequently, you must call universe_ready before re-entering the simulation loop if any new graphical entities have been created.

universe_go

```
void universe_go();
```

universe_go starts the main simulation loop. Control does not return to the statement following the call to universe_go until the universe_stop function is called. However, your application can gain control through action functions. The universe has a user-specifiable action function (set by calling universe_setactions) which is invoked before the rendering occurs for each frame. Individual objects can also have task functions which are called for the object once per frame (see object_settask). The principle is similar to the "callback" or "event" functions typically provided by a window management system.

Before calling universe_go for the first time you must call universe_ready. You must also call universe_ready before subsequent

The second argument modelview is a pointer to a 6D position/orientation record (a Posn6d) in which viewpoint information read from the file is returned. Currently, only DXF files and files which have been created with universe_save contain information about the viewpoint in effect. Other files return a default position and orientation (at the origin (0,0,0) and looking down the +Z axis). The modelview information is useful for recreating the viewpoint with which a model was saved, as in the Walkthrough sample code in Appendix B.

The final argument to universe_load is a scale factor which is applied to the geometry read in from the file. The entire model is scaled by this factor about the world coordinate frame origin. If you do not wish to scale the model, pass in a scale factor of 1.0. See the section called "Roundoff and scaling" in Chapter 11 for some issues related to universe scaling.

Scaling your model up or down too much is not desirable. As a rule of thumb, you should not scale by more than 2 orders of magnitude, i.e. 100.0 or 0.01. If you need to scale more than this, perform the scaling in your modelling program prior to saving out the geometry file to be loaded into WorldToolKit. Scaling up by too large a factor will impede the rendering performance of WorldToolKit. Scaling down by too small a factor can cause polygons to vanish.

When universe_load is called, it takes time to load in a new model and establish the necessary data structures. It is best not to call universe_load from within the simulation loop when real-time performance is required. universe_load is typically called either from outside the simulation loop (i.e. before universe_go has been called, or after a call to universe_stop), or from an interface within which the end user is aware that a new model is being loaded in.

After any call to universe_load and before entering the simulation, you must call universe_ready to establish colortable settings.

The universe_load function returns TRUE if the universe loading was successful, or FALSE if it failed for any reason.

Successful completion of the call to universe_load will trigger execution of any entry action function established by prior call to universe_set_entryfn.

universe_save

```
FLAG universe_save(filename)
    char *filename;
```

Use universe_save to save out the geometry and data structures that were originally created by universe_load. The file that is created by universe_save can be passed in to universe_load at any later time to quickly reload this stationary geometry.

universe_save is very useful as a preprocessor for geometry which you would like to load as quickly as possible into your application. A stand alone program which uses universe_load and universe_save to create preprocessed files is given as a sample application in Appendix B.

universe_save saves out the current position and orientation of the universe viewpoint. This viewpoint information is read back in when the file is subsequently loaded in with universe_load.

Currently universe_save will save out your stationary background only if this stationary geometry is the only graphics in the universe at the time universe_save is called, and there are no moving objects present in the universe. The function returns FALSE if it was unable to save your universe for any reason.

Simulation management

The simulation loop is the heart of a WorldToolKit application. The simulation loop is entered by calling universe_go and is exited by calling universe_stop. Alternatively, you can use the function

Introduction

What is WorldToolKit...

WorldToolKit is a library of functions written in standard C that you can use to construct real-time 3D graphical applications. WorldToolKit is so named because your applications can resemble virtual worlds, where graphical objects can have real world properties and behavior. You can control these worlds with a variety of input sensors, from a simple mouse to a position-tracked 6D sensor. You can experience your world through the computer display (think of it as a movable window into your world) or by using a position-tracked, head-mounted stereoscopic display.

WorldToolKit is structured in an object-oriented way, although it does not use inheritance or dynamic binding. Most WorldToolKit functions are object-oriented in their naming convention, and are grouped into classes. These classes include "universe", "object", "sensor", "viewpoint", "light source", "portal", and "animation sequence". The lower level functions in the math library are not object-oriented.

In a WorldToolKit application, only one universe object exists. It is the container for all other objects that you add to it. For convenience, these objects are automatically added to the universe container when you create them. Since the simulation manager in WorldToolKit acts upon the objects contained in the universe container, you can tune your application by specifically removing unnecessary objects from the universe when you don't need them and adding them back when you do. Also, you can create all of your objects at the beginning of the application (which adds them to the universe) and then remove the ones that you will add back in at a later, more appropriate time.

object_addsensor() for graphical objects and viewpoint_addsensor() for viewpoints. However, it is an error to make a call such as:

```
object_addsensor(light,sensor)
```

with light declared as above, since the methods for graphics objects with names starting with "object_" expect a variable of type Object* as their first argument.

The WTK program

After you have installed WorldToolKit, your demo subdirectory will contain a program called WTK.EXE. WTK was written by Sense8 software engineers as an example of a typical WorldToolKit application. In its own right, WTK is a fun and useful program, however it is not intended to be a robust application. We hope that you will enjoy using it.

WTK is a complete interactive "walk-through" program that has the following features:

1. monoscopic or stereoscopic viewing.
2. Polhemus, Spaceball, CiS Geometry Ball Jr. and mouse support for manipulating the viewpoint.
3. import of models and "parts" from DXF and other file formats.
4. interactive texturing and object manipulation.
5. saving of your new virtual world to disk.

Before running WTK, make sure that you have installed WorldToolKit and your DVI™ software and hardware correctly. Try running the test programs distributed with your ActionMedia™ 750 Delivery Board to

confirm that your PC is correctly configured and your software is correct. More complete installation instructions are provided in the WorldToolKit Installation Guide.

Running WTK

The command line argument to run WTK is:

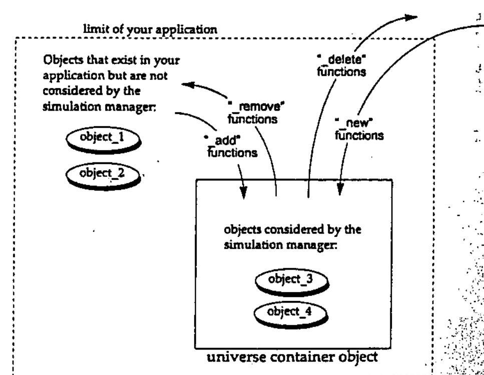
```
WTK [-xn] [-dparts_dir] model_name
```

where

- *x* — can be either s (for Spaceball), g (for GeoBall) or p (for Polhemus). If this flag is not included, then WTK assumes that you wish to use the mouse to move the viewpoint.
- *n* — is either 1 or 2, for the COM port that your input sensor is attached to.
- parts_dir — is optional and is the name of the directory that contains one or more objects that you would like to load into the virtual world with the stationary object. The flag -d is required to preceed the parts_dir.
- **model_name** — **is the path\name of the model** that you would like to load in as the "stationary" or background object, and should be the name of a *.DXF file, *.GEO file, or *.NFF file.

WTK will read in this file, create a 3D model out of it and launch you into your virtual world. WTK will also look in the directory that you optionally specify as parts_dir and will load any geometry files found in that directory that WTK knows how to read. For example, if you have five files in parts_dir (e.g. table.dxf, chair.dxf, book.dxf, test.c and

Naming conventions for WorldToolKit functions are such that each class of object has a typedef defining a *handle* for an object of that type. For instance, Sensor is a pointer to a sensor object, and Serial is a pointer to a serial port object. The state of any object must be accessed through "set" and "get" access methods defined in the WorldToolKit library. Objects are always dealt with through their handles. In fact, the internal state of WorldToolKit objects is not accessible except through WorldToolKit method calls provided for this purpose. Objects in WorldToolKit are "opaque", enforcing data abstraction.



All methods (functions) acting on a given class have by convention a name which begins with the class name. In addition, all classes accessible by the user have an object constructor whose name ends in "_new" which returns a new object of the given class, and an object destructor ending in "_delete" which accepts and destroys an object of the given class. For instance, the method:

```
Viewpoint *viewpoint_new()
```

creates and returns a new viewpoint object, as in:

```
newview = viewpoint_new();
```

This new viewpoint could subsequently be destroyed by the call:

```
viewpoint_delete(newview);
```

Finally, all methods expect a handle to an object of their class as the first argument. This is the object to which the method is directed. To copy an object, one would call the function object_copy, which takes a handle to an already-existing object and returns a handle to a newly-created copy of that object:

```
Object *old_object, *new_object;
new_object = object_copy(old_object)
```

An exception to this is constructor methods, which do not take a handle to an object (because one does not yet exist!) but instead return such a handle after creating the object.

The universe object is special in that there can be only one universe at any given time. It does not follow the convention that its methods require a universe handle as the first argument.

Note that an object should not be supplied as the first argument for a method which is intended for another class, as tempting as this might be. For instance, if we have a light object declared as Light *light, then to add a sensor to the light, the light_addsensor method should be used:

```
Sensor *sensor;
light_addsensor(light, sensor);
```

There are other "addsensor" methods in WorldToolKit, such as

calls to universe_go if new graphical entities have been created since the last call to universe_ready. See the above description of universe_ready.

universe_go1

```
void universe_go1();
```

For certain applications, it may not be desirable to relinquish control of the simulation to the WorldToolKit simulation manager and rely solely on callback functions such as the universe's action function, or object task functions. For applications that have to do something once per frame before and/or after the WorldToolKit simulation manager performs its rendering, it is convenient to use the universe_go1 function. This function goes through the simulation loop once each time it is called.

An example of such an application may be a telerobotic simulation, where devices in the real world must have their state queried and this state is used to update the graphics displayed by WorldToolKit. For such an application, the main loop may look like:

```
universe_ready();
while (TRUE) {
    /* query the state of sensors in the real world */
    ....        /* application specific code goes here */
    universe_go1();     /* draw 1 frame */
    ....        /* more application specific code can go here */
}
universe_delete();
```

Note that in this example, your application code is responsible for exiting from the main loop.

Before calling universe_go1 for the first time, universe_ready must be called. You must call universe_ready before subsequent calls to

universe_go1 if new graphical entities have been created since the last call to universe_ready. See the previous description of universe_ready.

universe_stop

```
void universe_stop();
```

Call universe_stop to exit the simulation loop entered by calling universe_go. Typically universe_stop is called from the universe's action function. An example of such an action function is the following, where geoball is a pointer to a sensor object which is assumed to have been allocated in the application code. For a more complete example, see the Walkthrough application sample code in Appendix B.

```
/* Exit the simulation loop if the left geoball button has been pressed. */
void actions()
{
    if (.sensor_getmiscdata(geoball) & GEOBALL_LEFTBUTTON ) {
        universe_stop();
    }
}
```

universe_vacuum

```
void universe_vacuum();
```

universe_vacuum empties the universe of all graphical entities which have been created without affecting the state of the windowing system or graphics device. Specifically, universe_vacuum frees all objects of type Object, Animation, and Portal, as well as graphical entities created with universe_load.

In addition, all structures allocated in support of the above objects are also freed, including the bitmaps which were allocated for any textures in use.

Note, however, that sensor objects and viewpoint objects (including the universe's current viewpoint) are unaffected by a call to universe_vacuum. For convenience, light objects are also preserved unless explicitly destroyed.

Note that universe_vacuum is called automatically by universe_load, to ensure that all graphical structures have been freed before new ones are created. Therefore, any calls to functions which create graphical objects, animation sequences, or portals must occur after universe_load is called.

The universe action function

The universe's action function is a user-defined function which is called by the simulation manager each time through the simulation loop. Figure 1.1 shows where the action function is called with respect to the other events in the simulation loop.

Action functions for the universe and the objects it contains define and control activity in the simulation. In the action function, events involving any WorldToolKit objects, graphical or otherwise, can be specified. Some examples of events which might be specified in the universe action function are:

1. Program termination by having a button press trigger a call to universe_stop.
2. Simulation of changing lighting conditions with calls to light_setposition or light_setintensity.
3. Testing for intersections of graphical objects using object_intersect or universe_intersect, and specification of what happens when intersections occur.
4. Event handling for a user interface. The action function might call universe_pickobject and then specify what is to be done with a selected object.

Actions pertaining to a specific graphical object can be specified in the object's task function, which is set with object_settask.

universe_setactions

```
void universe_setactions(actionfn)
void (*actionfn) ();
```

The universe's action function is a user-defined function which is called once each time through the simulation loop. universe_setactions sets this function. See the sample application code in the Appendix B for examples of useful action functions.

The universe's viewpoint

A viewpoint object stores parameters such as position, orientation, and viewing angle which define the way in which images are drawn. Further information on viewpoints can be found in Chapter 6. When the universe is constructed (with the call universe_new), a viewpoint object is automatically created for the universe. For many applications, this one viewpoint object will be sufficient. For others, it may be convenient to be able to switch back and forth between different viewpoints from which the universe is rendered. The function universe_setviewpoint lets you do just that.

universe_setviewpoint

```
void universe_setviewpoint(view)
Viewpoint *view;
```

universe_setviewpoint sets the universe's current viewpoint, that is, the viewpoint used to render the virtual world. In the following example,

a new viewpoint is created with the same orientation as the current viewpoint, but "zoomed out" away from this viewpoint so that most of the universe is in view.

```
Viewpoint *newview;     /* new viewpoint for "zoomed" view */
Posn3d dir;             /* viewpoint direction */
Posn3d midpt;           /* universe midpoint */
Posn3d newpos;          /* new viewpoint 6d position */

/* make a new viewpoint by copying the universe viewpoint */
newview = viewpoint_copy(universe_getviewpoint());

/* get direction of universe's current viewpoint */
viewpoint_getdirection(newview, dir);

/* scale this unit vector by universe radius */
mult_sv(universe_getradius(), dir);

/* move the new viewpoint out from universe midpoint
along this vector */
universe_getmidpoint(midpt);
subtract(midpt, dir, newpos);
viewpoint_setposition(newview, newpos);

/* finally, switch to this new viewpoint */
universe_setviewpoint(newview);
```

universe_getviewpoint

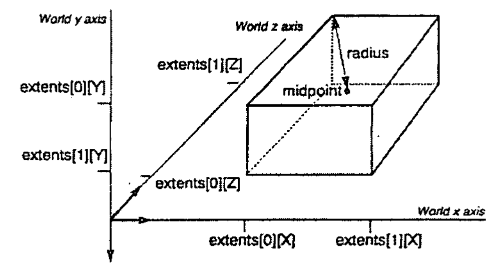
```
Viewpoint *universe_getviewpoint();
```

universe_getviewpoint returns a pointer to the universe's current viewpoint, that is, the viewpoint from which the universe is currently being rendered.

Universe geometrical properties

WorldToolKit functions provide access to three useful geometrical parameters describing the graphical entities in the universe taken as a whole. These are the extent of these entities in the world coordinate frame, the midpoint of these extents, and the "radius" of this extents box. Figure 1.2 illustrates these parameters.

Note that these parameters can be retrieved with WorldToolKit calls, but they can not be directly set. Instead, these parameters are determined in the function universe_ready from the graphical entities in the universe at the time the call is made.



The extents box is the smallest world-coordinate frame aligned box which fits about the graphical entities in the universe. The radius is the distance from the midpoint of the extents box to one of its corners.

Figure 1.2 Universe geometric parameters: extents, radius and midpoint.

universe_getextents

```
void universe_getextents(extents)
Posn3d extents[2];
```

The universe's extents are the minimum and maximum world coordinate values of all graphical entities. These values are calculated in the function universe_ready. universe_getextents puts the minimum X, Y, and Z values in world coordinates of graphical entities into the vector extents[0], and puts the maximum such values into extents[1].

One use for the universe's extents is in restricting viewpoint motion. Your universe action function might look to see whether the viewpoint was within the universe's extents, and if not, call viewpoint_move to ensure that the viewpoint stayed within the spatial extents of the graphics in the universe.

universe_getradius

```
float universe_getradius();
```

The universe's radius is the distance from the midpoint of the universe's "extents box" to one of its corners (see Figure 1.2). universe_getradius returns this value.

It is often useful to scale distances in an application (for example, the velocities of moving objects or viewpoints) with a characteristic distance scale in the universe. The universe's radius is convenient for that purpose. If sensor sensitivity is scaled as in the example below, then any object that the sensor is attached to will move a distance proportional to the universe's radius each time through the simulation loop. Based on the way the sensor_setsensitivity function operates, it

will take 200 frames at maximum sensor displacement to move the sensor position from one extreme of the universe to the other.

```
Sensor *sensor;

/* scale sensor sensitivity with the size of the universe */
sensor_setsensitivity(sensor, 0.01 * universe_getradius());
```

universe_getmidpoint

```
void universe_getmidpoint(p)
Posn3d p;
```

The universe's midpoint is the midpoint of the universe's world-coordinate aligned extents box (see Figure 1.2). universe_getmidpoint stores this point in p.

Accessing objects in the universe

The universe's "objects" may include graphical objects, sensors, lights, and animation sequences. (The universe's viewpoint object, of which there is only one at any given time, is discussed separately above.) The following functions provide access to the list of each type of object currently in the universe. To iterate through this list, the corresponding iterator functions object_next, sensor_next, light_next, or animation_next are used.

universe_getobjects

```
Object *universe_getobjects();
```

universe_getobjects returns a pointer to a list of graphical objects that are currently in the simulation. This list consists of all of the objects which have been created with object_new and which have not been removed from the universe with object_remove or object_delete. Also included in the list are the current "keyframe" objects for animation sequences in the universe.

To iterate through this list of objects, use the function object_next, as in the following example:

```
Object *object;
Posn3d p;

/* iterate through the universe's object list,
making all objects 10 times larger */
for ( object=universe_getobjects() ; object ;
            object=object_next(object) ) {

    /* scale the object about its midpoint */
    object_getposition(object, p);
    object_scale(object, 10.0, p);
}
```

universe_getsensors

```
Sensor *universe_getsensors();
```

universe_getsensors returns a pointer to a list of all sensors currently in the universe. Use the function sensor_next to iterate through this list.

universe_getlights

```
Light *universe_getlights();
```

universe_getlights returns a pointer to a list of all lights currently in the universe. Use the function light_next to iterate through this list.

universe_getanimations

```
Animation *universe_getanimations();
```

universe_getanimations returns a pointer to a list of all animation sequences currently in the simulation. This includes all animation sequences that have been created with animation_new, and which have not been removed from the universe with either animation_remove or animation_delete. Use the function animation_next to iterate through this list.

Picking functions

With universe_pickpolygon and universe_pickobject you can select polygons and objects based on their projection into the 2D window or screen.

universe_pickpolygon

```
Poly *universe_pickpolygon(point)
Posn2d point;
```

universe_pickpolygon takes a 2D screen point and returns the front-most polygon at this point. This polygon can then be passed in to

functions such as poly_texture_apply to modify its appearance. If no polygons project to this screen point, then universe_pickpolygon returns NULL.

universe_pickpolygon returns NULL if the point passed in does not lie within the screen boundaries. This means that point[X] must be between 0 and Width-1, and point[Y] must be between 0 and Height-1.

Width is 256 in low resolution mode, 512 in high resolution mode. Height is 240 in low resolution mode, 480 in high resolution mode.

Keep in mind that Posn2d's are made up of floating point values, so that point[X] and point[Y] are floats.

One convenient way to obtain a screen point to supply to universe_pickpolygon is through the sensor_getrawdata function. The following code fragment returns the handle of the polygon under the mouse cursor:

```
Polygon *poly;
Sensor *mouse;
poly = universe_pickpolygon(*(Posn2d*)sensor_getrawdata(mouse));
```

universe_pickobject

```
Object *universe_pickobject(point)
Posn2d point;
```

universe_pickobject takes a 2D screen point and returns the front-most graphical object at this point. As an example, the following code fragment selects the object in the physical center of the screen. If no graphical object is at this screen point, then universe_pickobject returns NULL.

universe_pickobject returns NULL if the point passed in does not lie within the screen boundaries. This means that point[X] must be between 0 and Width-1, and point[Y] must be between 0 and Height-1.

Keep in mind that Posn2d's are made up of floating point values, so that point[X] and point[Y] are floats.

```
Posn2d point;
Object *object;

/* set point to midpoint of screen and find any object there */
point[X] = Width2; point[Y] = Height2;
object = universe_pickobject(point);
```

Testing for intersections

The following function tests for intersections of a dynamic object with any of the other graphical entities in the universe.

universe_intersect

```
FLAG universe_intersect(object)
Object *object;
```

universe_intersect determines whether an object's bounding box intersects any of the polygons of any graphical entities currently in the universe. (See Figure 2.1 for an illustration of an object's bounding box.) The graphical entities which are tested for intersection with the object's bounding box may have been created with universe_load, object_new, object_copy, animation_new, or the terrain functions. If an intersection is found, the function returns TRUE, otherwise it returns FALSE.

Because the object's bounding box (rather than polygons) is tested against the polygons of the other graphical entities, universe_intersect sometimes return TRUE even when there is no visible intersection of the object's surface with the other graphical entities.

The greater the number and complexity of graphical entities in the universe, the longer universe_intersect takes, since it tests for intersections with individual polygons. A less accurate but faster intersection test is provided by the function object_intersect.

The following is an example of an object task function which uses universe_intersect to prevent an object from intersecting the other graphical entities in the universe. This task function is assigned to the object using the function object_settask.

See also object_boundingbox.


```
/*
 * Each time through the simulation loop when this function is
 * called, the object's position and orientation are stored (in lastp
 * and lastq), so that next time, if the object is found to intersect
 * something in the universe, it can be restored to lastp and lastq.
 * The initialized FLAG becomes TRUE as soon as the object
 * is found to not intersect anything in the universe. This way,
 * when the object is restored to lastp and lastq, we know that
 * that location is one in which there is no intersection. */
 */
void avoidance_task(o)
    Object *o;
{
    static FLAG initialized = FALSE;
    static Posn3d lastp;
    static Quat lastq;

    /* if there is an intersection, move the object back to where
    there wasn't one. */
    if ( universe_intersect(o) ) {
        if ( initialized ) {
            object_setposition(o, lastp);
            object_setorientation(o, lastq);
        }
    }
    else {
        initialized = TRUE;
    }

    /* store position and orientation for next time */
    object_getposition(o, lastp);
    object_getorientation(o, lastq);
}
```

Graphical properties

universe_getbgcolor

```
short universe_getbgcolor();
```

universe_getbgcolor returns the background color used when the universe is rendered. This value is a short in the range [0x000, 0xfff]. In other words, the color is represented with 4 bits for each of R, G, and B. The default background color is given by the defined constant DEFAULT_BGCOLOR, which is 0x00f (blue).

universe_setbgcolor

```
void universe_setbgcolor(bgcolor)
short bgcolor;
```

universe_setbgcolor sets the background color used when the universe is rendered. bgcolor is a true color in the range [0x000, 0xfff], so that there are 4 bits for each of R, G, and B.

If you do not explicitly set a background color, the default background color is used. This color is the defined constant DEFAULT_BGCOLOR, which is 0x00f (blue).

Entering and exiting a universe

Through the use of portals (Chapter 8) it is possible to automatically switch universes during the course of running a WorldToolKit application when the universe's viewpoint crosses a given polygon.

Partitioning the environment into multiple universes connected by portals is useful because it increases rendering speed. Only the objects in the current universe are considered by WorldToolKit for rendering. Partitioning the universe is useful because it also simplifies the process of constructing large virtual environments. It means that you need to construct only a portion of the virtual world at a time, and that the task of building up the overall environment can be segmented among different people.

It may be necessary to cause a function to be invoked when a universe of a given name is entered or exited. For example, sensors may need to be scaled or configured differently for a new universe, or background color may need to be changed. To automate the process of invoking a function upon change of universe, the functions universe_set_entryfn and universe_set_exitfn are supplied.

Any sensors in existence for the current universe are maintained when a new universe is entered, unless explicitly changed by the application code. Lighting is also maintained.

universe_set_entryfn

```
void universe_set_entryfn(name,fn)
char *name;
void (*fn)();
```

universe_set_entryfn function establishes a function to be called when the universe is entered, either implicitly through crossing a portal connected to that universe, or explicitly by universe_load.

The arguments to universe_set_entryfn are the name of a universe and a pointer to a function. The name of the universe is supplied as the first argument to the universe_load function. The action function can be any user-supplied function which expects no arguments and returns void.

For example, the call:

```
universe_set_entryfn("DOME", restrict_translation);
```

ensures that a user-supplied function void restrict_translation() will be called when the universe called "DOME" is entered.

To remove a universe entry function which has previously been specified, another call to universe_set_entryfn should be made, with a second parameter a NULL pointer appropriately cast as a pointer to a function returning void.

For example:

```
universe_set_entryfn("DOME", (void (*)()) NULL);
```

can be used to remove a previously established entry function for the universe named "DOME".

If the first argument to universe_set_entryfn is given as a blank character string ("") then the action function is taken as a generic action function, to be called when any universe is entered. If a universe has an action function and a generic action function has also been set, upon entering the universe the generic function is called first, then the action function specific to the universe is called.

universe_get_entryfn

```
PFV universe_get_entryfn(name)
char *name;
```

universe_get_entryfn function returns a pointer to the entry function for the universe with name name. The entry function is the function that was set using universe_set_entryfn. If no entry function was set for the universe, NULL is returned.

PFV stands for "pointer to function returning void", and is typedefed as:

```
typedef void (*PFV)();
```

universe_set_exitfn

```
void universe_set_exitfn(name, fn)
char *name;
void (*fn)();
```

universe_set_exitfn establishes a function which is invoked when the named universe is exited, either implicitly by entering another universe, or explicitly by program termination.

To remove a universe exit function which has previously been specified, another call to universe_set_exitfn should be made, with a second parameter a NULL pointer appropriately cast, as described previously.

If the first argument to universe_set_exitfn is given as a blank character string ("") then the action function is taken as a generic action function, to be called when any universe is exited. If a universe has an action function and a generic action function has also been set, upon exiting the universe the action function specific to the universe is called, then the generic function is called.

universe_get_exitfn

```
PFV universe_get_exitfn(name)
char *name;
```

universe_get_exitfn function returns a pointer to the exit function for

the universe with name name. The exit function is the function that was set using universe_set_exitfn. If no exit function was set for the universe, NULL is returned.

PFV stands for "pointer to function returning void", and is typedefed as:

```
typedef void (*PFV)();
```

Performance / Statistics functions

For optimizing performance, it may be necessary for application code to know how fast the simulation is running and how many polygons are present in the universe. Although at least the former of these could easily be written outside of WorldToolKit by making calls to system timer functions, the need for such functions is a common enough occurrence that they are provided as part of the library.

universe_framerate

```
float universe_framerate()
```

This function returns the number of frames per second at which the simulation is currently running. The number returned is actually a running average of the frame rate of the proceeding several frames, in an attempt to stabilize the reading to at least one decimal digit.

universe_npolygons

```
long universe_npolygons()
```

This function returns the total number of polygons for all graphical entities currently in the universe. See also object_npolygons, for a count of the number of polygons in a single graphical object.

universe_setresolution

```
void universe_setresolution(res)
FLAG res;
```

As described in Chapter 12, the WorldToolKit default is to use an adaptive resolution scheme which increases screen resolution when the display scene is not changing. Rendering at lower resolution is faster, rendering at higher resolution creates spatially smoother images.

Since there may be times when application code wishes to defeat the adaptive resolution scheme, the universe_setresolution function is provided. It takes an argument which should be one of the defined constants:

1. RESOLUTION_LOW - to force resolution to always be low.
2. RESOLUTION_HIGH - to force resolution to always be high.
3. RESOLUTION_ADAPTIVE - to return to adaptive resolution.

2

Graphical Objects

Introduction to the graphical object class

Graphical objects (or simply objects, for short) are the fundamental building blocks of a WorldToolKit application. These objects are graphical entities with which the user can interact and which can also interact with each other. They can be hierarchically organized, their motion or state can be affected by sensors, and they can have tasks assigned to them defining their functionality or behavior in the simulation.

Objects originate from a file describing their geometry and attributes. For example, this file could have been created by a CAD program and written in DXF format. To create a graphical object, the function object_new is called, passing in the filename of a 3D database describing the object to be created.

You can associate your own data with an object by using the object_setdata function which sets a void * pointer field in the graphical object struct provided for this purpose. This is described in the section called "User-specifiable object data".

Also, several types of terrain object construction functions are provided, making it easy to create virtual world landscapes.

Creating objects in your modelling program

Using a CAD program you can create a graphical scene for your WorldToolKit application in which the various graphical entities (dynamic objects and stationary backdrop) have the desired spatial relationships. One technique for accomplishing this is to initially build all of the geometry into one CAD file, positioning the various entities as desired. Then save out each portion of the model from which you wish to create a dynamic object or the stationary backdrop into a separate file.

For example, let's say that you want to create an office model that consists of office walls, a desk, a chair, and a book on the desk. And let's say that only the chair and the book will be movable (dynamic) objects. You would construct the model containing all of these components and save it out to a file. Then to create the file which contains the stationary backdrop (which will be passed to universe_load), start from the original file, erase the book and the chair, and save to file the resulting model which contains just the walls and the desk. Similarly, to create the file for the chair (which will be passed in to object_new), load in the original file, erase the walls, desk, and book, and save the result to a separate file. Similarly you can create the file from which the book object will be constructed.

If you are using AutoCAD, then another approach is to create each graphical object which you wish to load in separately to WorldToolKit on a separate layer. Once your model is constructed, you can successively for each object turn off all layers but the one that the object is on and save out each file.

An item deserving special mention is backface rejection. Most geometrical entities in the AutoCAD DXF standard are 2 1/2D entities, planar curves with extrusions. When these curves are "closed" it is possible for WorldToolKit to unambiguously interpret them in 3D as solids, and know which polygons are seen from their "inside" and which from their "outside". In such cases, in the interest of rendering

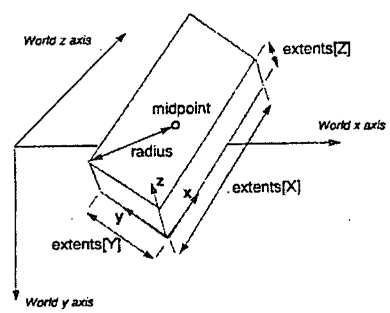
efficiency, backfaces are rejected at an early stage of the rendering pipeline. The result is that when you go inside closed solid objects, they will disappear!

If it is desired to have the inside of AutoCAD-modelled objects appear, then the models must be constructed so backfaces are not rejected. To guarantee the retention of backfaces, objects should be constructed of individual 3D polygons, extruded open polylines, or polyface meshes which are not closed in both directions.

Geometrical properties

Many things that you might wish to do with graphical objects may involve the object's geometrical properties. The geometrical properties of an object which can be accessed with WorldToolKit are the object's midpoint, radius, orientation, local coordinate frame axes, extents of an object-aligned bounding box, and pivot point. These parameters are defined below and illustrated in Figure 2.1.

WorldToolKit uses a right-hand rule world coordinate system, with the screen in X-Y space with X increasing left to right, Y increasing top to bottom, and the positive Z axis pointing into the screen.



The object's X axis typically coincides with the object's longest dimension, while its Y and Z axes coincide with its next longest and shortest dimensions, respectively.

Figure 2.1. Geometrical parameters of graphical objects.

An object's geometrical properties are computed in object_new once the 3D model geometry has been read in. These parameters are determined as follows:

1. **orientation.** This is computed using a principal axis calculation based on the object's vertices. Typically this results in an orientation for the object in which the longest dimension of the object is the X axis, the next longest dimension is the Y axis, and the shortest dimension is the Z axis. Orientation is stored as a quaternion.
2. **coordinate frame axes.** These are the X, Y, and Z axes of the object's local coordinate frame corresponding to the object's orientation.
3. **extents.** The extents of an object are the dimensions of an imaginary extents "box" about the object. This extents box is defined as the

smallest rectangular box that encloses all the vertices of the object and that is aligned with the local coordinate axes of the object.

4. **midpoint.** The midpoint of an object is the midpoint of the object's coordinate-aligned extents box defined above.
5. **radius.** The radius is the distance from the midpoint to a corner of the object-aligned extents box defined above.
6. **pivot.** An object's pivot point is the point about which the object rotates when object_rotate or object_move is called, or when a sensor attached to it generates rotational input.

Definition of reference frames for object motion

Many of the functions which let you move graphical objects within the virtual world take as an argument the reference frame in which the motion is to occur. These reference frames are illustrated in Figure 2.2.

1. FRAME_WORLD is the world coordinate frame. It is independent of the objects in the universe and is fixed in space.
2. FRAME_LOCAL is the local coordinate frame of the object determined from the location of the object's vertices when the object is constructed.
3. FRAME_VPOINT is the reference frame of the universe's viewpoint. If, for example, you wish to move an object in the direction that the viewpoint is looking, move it in the positive Z direction in FRAME_VPOINT.

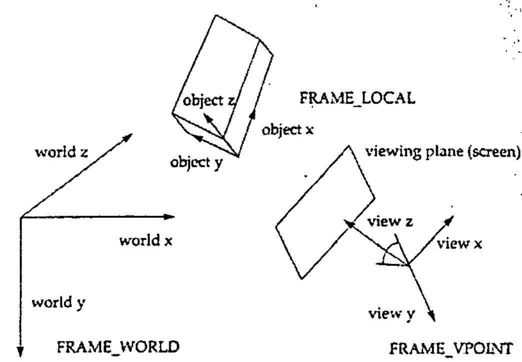


Figure 2.2 Reference frames for object motion.

Basic object management

This section describes the functions for creating and destroying objects, adding and removing objects from the simulation, copying objects, and saving objects from WorldToolKit out to file.

When an object is created in WorldToolKit, it is automatically added to the universe, becoming part of the simulation. As part of the simulation, the object is rendered (when within view), it performs its associated task, it interacts with other objects, and it responds to input from any sensors attached to it. You may, however, wish to create objects that are not always in the simulation. To avoid the overhead of loading an object in from file in the middle of the real-time loop, the

functions object_add and object_remove are provided. These functions let you quickly add or remove objects to or from the simulation at any time.

object_new

```
Object *object_new(filename, model6d, scale, fast)
char *filename;
Posn6d *model6d;
float scale;
FLAG fast;
```

object_new creates a graphical object from a file containing a description of 3D geometry. This file may be in DXF format, in WorldToolKit's neutral file format, or it may be a file generated by the Amiga programs VideoScape or Caligari. Or, it may be a binary file created with the function object_save.

Once object_new has created the object's polygons, it then computes the geometrical properties illustrated in Figure 2.1. The object is then added to the universe. Until the object is removed from the universe with object_remove or object_delete, it is part of the simulation.

Any calls to object_new must occur after the function universe_load is called (if universe_load is called by your program). This is because universe_load automatically calls universe_vacuum to make way for new graphics being loaded in. In addition, after all calls to object_new are made, and before calling universe_go, you must call universe_ready to establish the colortable.

Since it takes time to load in a new model, it is best to call object_new outside of the main simulation loop (i.e. before calling universe_go). If a graphical object is not needed at the start of the simulation, you can call object_remove to remove the object from the simulation, and then object_add when the object is needed.

filename is the name of a file containing the 3D geometry from which the object is constructed. For example, it could be the name of a DXF file.

The DOS environment variable WTMODELS gives a path to geometry files which is searched before the current directory for the filename passed in to object_new. For example, let's say that from the DOS command line you:

```
set WTMODELS=c:\dxf;c:\demo
```

Then, if object_new is called with filename "oplan.dxf", this file would first be searched for in c:\dxf. If not found there, it would be searched for in c:\demo. Finally, if still not found, it would be searched for in the current directory.

The argument **model6d** is a pointer to a Posn6d in which object_new puts the viewpoint position and orientation which may be contained in the model file. Certain 3D model database formats contain the viewpoint position and orientation with which the model was saved. Currently, the only file types which contain this information are DXF files and files created with object_save. In the latter case, the Posn6d stored is whatever Posn6d is passed in when the object_save call is made. If the object has just been read in from a DXF file, then the 6D viewpoint information read in could be passed to object_save. Note that because the Posn6d is a struct, you must pass in a pointer to the Posn6d for values to be returned in it. Other mathematical quantities in WorldToolKit, such as Posn3d and Quat, are arrays of floats and are thus not passed by pointer. Check the function prototypes carefully, and/or examine the include/mathlib.h file where the WorldToolKit mathematical types are defined.

scale is the factor by which the object is scaled about the world coordinate frame origin when it is loaded in. If you do not wish the object to be scaled, then this argument should equal 1.0. See the section called "Roundoff and scaling" in Chapter 11 for a discussion of issues related to the scaling of objects. The important aspect to keep in mind is that it is best not to scale an object by a factor that is any larger than

necessary, as described in the documentation for universe_load. See the functions object_scale and object_stretch for scaling and stretching the object about an arbitrary point.

fast is a FLAG which determines the method by which the object is rendered. Normally this FLAG should be FALSE. However, moving a complex object can be computationally intensive. If you find that your frame-rate is suffering as you move an object, then you may choose to set this FLAG to TRUE. A less computationally intensive rendering algorithm will then be used to display the object. The price for faster rendering may be an occasional rendering error.

The frame-rate increase that is obtained when fast is TRUE depends on many factors, including the complexity of the models being drawn, their spatial relationships, and how much other processing is going on, as requested by the universe's action function or object task functions. It may be possible to obtain a frame-rate increase of a factor of as much as 5. However, this is largely an experimental issue, and you will need to try it and see what works best for your particular application.

The object_new function returns a pointer to the newly created and loaded object if successful, otherwise (such as if the file was not found) it returns NULL.

The following code fragment is an example of using object_new:

```
Object *object;
Posn6d modelview;

/* create an object from "myfile"
   viewpoint position returned as modelview */
object = object_new("myfile", &modelview, 1.0, FALSE);
```

object_delete

```
void object_delete(object)
Object *object;
```

object_delete removes an object from the universe and frees its memory.

object_remove

```
void object_remove(object)
Object *object;
```

object_remove removes an object from the universe so that it is no longer a part of the simulation. The object can be put back into the universe by calling object_add. Until then, the object is not drawn, it does not perform its task, it does not interact with other objects, and it is not affected by sensors.

object_add

```
void object_add(object)
Object *object;
```

When an object is added to the universe with object_add, it becomes part of the simulation. When an object is in the simulation, it performs its task, it is affected by any sensors that are attached to it, and it is drawn when in view.

It is unnecessary to call object_add when an object is constructed with object_new, because object_new automatically adds it to the simulation.

object_save

```
FLAG object_save(object, filename, modelview)
Object *object;
char *filename;
Posn6d *modelview;
```

object_save saves an object's geometrical, color, and texture description to a binary format file. This object can later be recreated by passing in the name of this file (filename) to the function object_new.

object_save is useful whenever you need to store out the current graphical and geometrical state of an object. For example, say that you wish to create a simplified version of an object using the level-of-detail tool, and then use this object in an application. Since the level-of-detail computation can take some time, it would be best to load in the object from the original file, then call object_levelofdetail, and then save the object out using object_save. Your application would then call object_new with the name of the file saved out to load in the simplified object quickly.

object_save can also be useful for preprocessing complex CAD models, because for some models it will be faster to load the model in from the file saved by object_save than from the original CAD file.

object_save also saves out the position and orientation struct modelview that you pass in by address. This is useful, for example, for storing out the current viewpoint with the object. This information is recovered when the file created with object_save is passed in to object_new.

object_save saves out only the object passed in, and not any parent or child objects that the object may be hierarchically attached to. No information about object hierarchy is saved out. In addition, object_save does not save the object's task function, or data that may have been stored with the object using object_setdata, or any sensor information. Any portals associated with the object's polygons are also not saved.

See also object_new and universe_save.

object_copy

```
Object *object_copy(object)
Object *object;
```

object_copy returns a copy of an object. The copy is not automatically added to the universe. If you want the copy added to the universe, you must call object_add.

The copied object is geometrically and graphically identical to the one passed in. For example, the copied object has the same textures applied to it, and has the same pivot point as the original object. However, object data that is NOT copied from the original object to the object copy is hierarchical configuration, sensor attachment, the object's sensor "frame" (i.e. the reference frame in which sensors move the object), the object's user-defined data (see object_getdata and object_setdata), and the task function.

```
Posn3d dp;
Object *copy, *original;

/* copy the original object */
copy = object_copy(original);

/* add it to the universe */
object_add(copy);

/* place copy 200 distance units below the original in world frame */
dp[X] = dp[Z] = 0.0; dp[Y] = 200.0;
object_translate(copy, dp, FRAME_WORLD);
```


object_next

```
Object *object_next(object)
Object *object;
```

Use object_next to iterate through the objects in the universe's object list. This list consists of all objects currently in the simulation. It does not include any objects for which either object_remove or object_delete have been called.

The following code fragment prints pointers to all pairs of intersecting objects in the universe:

```
Object *o1,*o2;
for ( o1 = universe_getobjects() ; o1 ; o1=object_next(o1) ) {
    for ( o2=universe_getobjects() ; o2 ; o2=object_next(o2) ) {
        if ( object_intersect(o1,o2) )
            printf("Object %x intersects object %x\n",o1,o2);
    }
}
```

Level of detail

The function object_levelofdetail takes a graphical object and returns a simplified version of this object. This is useful for objects which are to be seen from distances at which some features would be too small to be discerned. Simplified models are more easily processed and drawn, and so higher frame rates are obtained when they are used in place of complex ones. For this reason, it may sometimes be useful to use objects simplified with object_levelofdetail even when they are to be viewed up close.

object_levelofdetail

```
Object *object_levelofdetail(object, distance)
Object *object;
float distance;
```

object_levelofdetail takes a graphical object and returns a simplified object which is indistinguishable from the original when viewed from the specified distance and with the current viewing angle. object_levelofdetail does not automatically add the new object to the universe. Call object_add to add the simplified object to the simulation.

object_levelofdetail is passed a distance which determines how much the object is simplified. The greater the distance, the more the object is simplified. The simplification that occurs also depends on the viewing angle that is set at the time that object_levelofdetail is called. Therefore, before calling this function, you may want to first set the view angle with viewpoint_setviewangle.

object_levelofdetail does not work in real-time because the task of simplifying an object is computationally intensive. The amount of time that it takes object_levelofdetail to generate a simplified object depends on how much the object is simplified, which depends in turn on the distance that is passed in. (The greater the distance at which the object is to be viewed, the greater the number of features in the model which can be eliminated without being missed.) The following suggestions may help when using this function:

1. Precompute simplified objects by writing a program which calls object_levelofdetail and then saves the result to file using object_save.
2. Pass in a distance to object_levelofdetail which is a multiple of the object's radius. This will give you a better sense of how much simplification will occur, and therefore how long you can expect the level of detail computation to take. The following example computes a simplified model appropriate for a viewing distance of 8 times the radius of the object and the current view angle (which is automatically taken into account when object_levelofdetail is called).


```
Object *object, *simplified_object;
float radius;

radius = object_getradius(object);
simplified_object = object_levelofdetail(object, 8.0 * radius);
```

A typical way in which level of detail is used is in swapping a complex object for a simplified one when the viewpoint moves sufficiently far away from the object. The following function illustrates how this can be accomplished. Note that such a function could be called from the universe's action function. (See universe_setactions.)

```
void swap_objects(objcomplex, objsimple, distance)
    Object *objcomplex, *objsimple;
    float distance;   /* distance beyond which swap simple model in */
{
    Posn3d p;        /* object position */
    Posn3d pview;    /* viewpoint position */
    float d;         /* distance from viewpoint to object */

    /* start out looking at complex object */
    static FLAG complex = TRUE;

    /* get position of the object currently in the universe */
    if ( complex ) {
        object_getposition(objcomplex, p);
    }
    else {
        object_getposition(objsimple, p);
    }
    /* get viewpoint position */
    viewpoint_getposition(universe_getviewpoint(), pview);

    /* compute distance from viewpoint to object */
    d = distance(pview, p);
```

```
    /* if appropriate, swap the models. This example assumes that
    the models have the same position and orientation. */
    if ( complex && d>distance ) {
        object_remove(objcomplex);
        object_add(objsimple);
        complex = FALSE;
    }
    else if ( !complex && d<=distance ) {
        object_remove(objsimple);
        object_add(objcomplex);
        complex = TRUE;
    }
}
```

Positioning an object

object_setposition

```
void object_setposition(object, p)
Object *object;
Posn3d p;
```

object_setposition translates the object so that its midpoint coincides with p.

Unlike the call object_translate which moves the object by the (relative) Posn3d passed in, object_setposition moves the object to the absolute location in world coordinates of the Posn3d passed in.

object_getposition

```
void object_getposition(object, p)
Object *object;
Posn3d p;
```

object_getposition retrieves the midpoint of the object's object-aligned bounding box and stores it in p.

object_translate

```
void object_translate(object, p, frame)
Object *object;
Posn3d p;
short frame;
```

object_translate translates an object by a relative amount given by a vector p in the specified frame (FRAME_WORLD, FRAME_LOCAL, or FRAME_VPOINT).

In the following example, to simulate an automobile driving straight ahead, the object is translated in its local reference frame in the direction of its longest dimension. (This is the object's X axis, as shown in Figures 2.1 and 2.2). The amount that it translates is proportional to its actual extent in this dimension.

```
Object *automobile;
Posn3d translation, extents;

/* get the object's extents vector */
object_getextents(automobile, extents);

/* set translation vector proportional to length*/
translation[X] = 0.2 * extents[X];
translation[Y] = translation[Z] = 0.0;
```

```
/* translate along its longest dimension */
object_translate(automobile, translation, FRAME_LOCAL);
```

object_move

```
void object_move(object, change, frame)
Object *object;
Posn6d *change;
short frame;
```

object_move translates and rotates an object with one call. This function takes a pointer change to a Posn6d which specifies the amount by which the object is to be translated and rotated. In other words, this Posn6d contains the change in position and orientation applied to the object, rather than the absolute position and orientation to which the object is moved.

object_move moves the object in the reference frame that you specify. frame must be one of the defined constants FRAME_WORLD, FRAME_LOCAL, or FRAME_VPOINT. These reference frames are defined above and illustrated in Figure 2.2.

The rotational change passed in to object_move is applied about the object's pivot point. To rotate an object about a point other than its pivot point, use the function object_rotatepoint.

See also object_translate, object_rotate, object_setposition, and object_setorientation.

Orienting an object

object_setorientation

```
void object_setorientation(object, q)
Object *object;
Quat q;
```

object_setorientation rotates the object to the orientation given by the quaternion q.

Unlike the call object_rotate which rotates the object by the (relative) rotation passed in, object_setorientation rotates the object to the absolute Quat passed in.

See Chapter 11 for conversion functions which convert various representations of orientation into quaternion form.

object_getorientation

```
void object_getorientation(object, q)
Object *object;
Quat q;
```

object_getorientation stores the object's orientation in the quaternion q. The Math Library (Chapter 11) contains conversion functions for converting quaternions into other representations for orientation.

See also object_setorientation.

object_rotate

```
void object_rotate(object, axis, radians, frame);
Object *object;
short axis;
float radians;
short frame;
```

object_rotate rotates an object around a given axis about the object's pivot point (See Figure 2.1.). axis is one of the defined constants X, Y, or Z, and pertains to the reference frame that you specify (FRAME_WORLD, FRAME_LOCAL, or FRAME_VPOINT).

See also object_move.

object_rotatepoint

```
void object_rotatepoint(object, rotation, point, frame);
Object *object;
Quat rotation;
Posn3d point;
short frame;
```

object_rotatepoint rotates an object about a 3D point specified in world coordinates. The object is rotated with respect to the axes of the specified reference frame (FRAME_WORLD, FRAME_LOCAL, or FRAME_VPOINT) about this 3D point.

rotation is specified as a quaternion. If the orientations you prefer to work with are represented as matrices or euler angles, functions in the Math Library (Chapter 11) will be of use for conversion between these alternative representations.

object_setpivot

```
void object_setpivot(object, pivot)
Object *object;
Posn3d pivot;
```

object_setpivot sets the object's pivot point. An object's pivot point is the point about which the object rotates when **object_rotate** or **object_move** is called, or when a sensor attached to the object generates rotational input.

Pivot points are specified in world coordinates. An object's pivot point is automatically updated when an object moves, so that the object always pivots about the same point on or relative to the object.

Pivot points are particularly useful in object hierarchies, for which it is common to want a sub-object to rotate in a particular way with respect to the parent object. For example, in an object hierarchy which models a robot arm, object_setpivot would be used to define the location of the various joints.

An example of using an object's geometrical properties to define its pivot point is given in the section "Object hierarchies" in this chapter.

object_getpivot

```
void object_getpivot(object, pivot)
Object *object;
Posn3d pivot;
```

object_getpivot retrieves the object's pivot point and stores it in pivot. pivot is a world coordinate frame point. When an object is constructed, its pivot point is taken by default to coincide with its midpoint. To change the pivot point, use object_setpivot.

object_getaxis

```
void object_getaxis(object, dim, vector)
Object *object;
short dim;
Posn3d vector;
```

object_getaxis retrieves the unit vector corresponding to the object's X, Y, or Z axis (whichever is specified in dim) in vector. Figures 2.1 and 2.2 illustrate the object's local reference frame and define these axes.

object_alignaxis

```
void object_alignaxis(object, dim, dir)
Object *object;
short dim;
Posn3d dir;
```

object_alignaxis rotates an object about its pivot point so that the specified object axis (one of the defined constants X,Y, or Z) as given in dim aligns with, that is, points in the same direction as, the direction vector dir passed in. The following is an example of a function which takes a Light * and an Object * and aligns the graphical object with the light, both in position and direction.

```
void align_object_with_light(flashlight, light)
    Object *flashlight;   /* object looks like a flashlight */
    Light *light;
{
    Posn3d lightpos, lightdir;

    /* retrieve light position and move graphical flashlight there */
    light_getposition(light, lightpos);
    object_setposition(flashlight, lightpos);
```



```
    /* retrieve light direction, and align flashlight's
    longest dimension (X) in this direction. */
    light_getdirection(light, lightdir);
    object_alignaxis(flashlight, X, lightdir);
}
```

In this example, assuming the flashlight is rotationally symmetric about its longest dimension or axis, then it won't matter if the flashlight is twisted about this axis. It will look the same so long as the axis aligns with the light direction.

However if the object that you call object_alignaxis for is not rotationally symmetric about the specified axis, you should note the following. When this function is applied to an object, even though the object's specified axis ends up pointing in the specified direction, the object will be twisted by an arbitrary amount about that direction. To change the amount by which the object is twisted about this axis, you can use the function object_rotate. Your program might include the following lines:

```
Object *object;
Posn3d dir;

object_alignaxis(object, X, dir);
object_rotate(object, X, PI, FRAME_LOCAL);
```

In this example, the object's X axis is aligned with dir. Then, the object is rotated about its X axis by PI. Note the use of FRAME_LOCAL to rotate the object about its own X axis.

No translation is applied to the object by object_alignaxis.

object_alignwithworldaxes

```
void object_alignwithworldaxes(object)
Object *object;
```

object_alignwithworldaxes rotates an object (about its pivot point) so that the object's local coordinate frame has the same orientation as the world coordinate frame. (See Figure 2.2.) No translation is applied to the object.

Object extents and intersections

object_getradius

```
float object_getradius(object)
Object *object;
```

object_getradius gets the radius of the object. The object's radius (see Figure 2.1) is defined as the distance from the midpoint of the object to a corner of the object-aligned bounding box.

object_getextents

```
void object_getextents(object, extents)
Object *object;
Posn3d extents;
```

object_getextents retrieves the extents of the object's object-aligned bounding box and stores them in extents. extents is a Posn3d containing the X, Y, and Z dimensions of the bounding box, as illustrated in Figure 2.1.

object_intersect

```
FLAG object_intersect(object1,object2)
Object *object1, *object2;
```

object_intersect returns TRUE if the object-aligned bounding boxes of two objects intersect. Otherwise it returns FALSE. If object1 and object2 are pointers to the same object, object_intersect returns FALSE. In other words, objects do not intersect themselves.

For information on using object_intersect when one or more of the objects has been stretched, see the description under object_stretch.

See also the function universe_intersect.

object_boundingbox

```
Object *object_boundingbox(object, width)
Object *object;
float width;
```

object_boundingbox returns a graphical object which is a polygonal representation of the object-aligned bounding box of the graphical object passed in. Creating a bounding box object is a convenient way to highlight a graphical object in the virtual world. For example, your program may allow users to select objects in the space (such as with universe_pickobject) and then modify them in some way. Bounding boxes can be useful in such a program for showing which object has been selected.

The bounding box object which is created consists of one polygon for each edge of the bounding box of the original graphical object. (See Figure 2.1.) The width of each polygon is the value width passed in. It may be useful to pass in a width which is proportional to a characteristic distance scale of the object such as its radius. Each polygon makes a 45 degree angle to the two bounding box faces which

meet at the edge, so that edges are visible even when the object is viewed head on. Bounding box polygons are white.

When a bounding box object is created, its pivot point is set to that of the original object so that the bounding box rotates about the same point as the original object.

object_boundingbox calls object_add to automatically add this "bounding box" object to the universe. Once created, the bounding box object is a graphical object like any other. From that point, it has no association with the original object unless you explicitly program it to have one.

For example, if you want the bounding box to move with the object it bounds as the original object moves, then you must attach any sensors which are attached to the original object to the bounding box object, and call any object move functions called for the original object also for the bounding box object.

Another use for bounding boxes is as a wireframe representation of an object. Let us say that your application lets users select an object an then interactively (for example with a sensor) relocate the object. The application will have a higher frame-rate if the user interactively places that bounding box object instead of the original object. Once the bounding box had been moved to the desired location, then the original object could be moved. The following example illustrates this:

```
Object *object;    /* the object we want to move */
Object *bbox;      /* bounding box for the original object */
Sensor *sensor;    /* for moving the bounding box */
Posn3d p;          /* final position of bbox */
Quat q;            /* final orientation of bbox */

/* highlight the object by constructing a bounding box object for it.
Make the width of the bounding box polygons proportional to the
object radius. */
bbox = object_boundingbox(object, 0.05*object_getradius(object));
```



```
/* attach the sensor to the bbox. move it in the viewpoint frame. */
object_addsensor(bbox, sensor, FRAME_VPOINT);

/* ......the user moves the bbox to a new location and indicates
that they are done..... */

/* the object is relocated to the location of the bounding box. */
object_getposition(bbox, p);
object_setposition(object, p);
object_getorientation(bbox, q);
object_setorientation(object, q);

/* un-highlight the object by deleting the bounding box. */
object_delete(bbox);
```

See the function object_stretch for information about calling object_boundingbox for an object which has been stretched.

Scaling and stretching

object_scale

```
void object_scale(object, factor, point)
Object *object;
float factor;
Posn3d point;
```

object_scale scales an object by a specified factor about a specified point in the world coordinate frame. If the scale factor equals 1.0, then object_scale has no effect.

In the following example, an object is scaled by a factor of 2 about its midpoint.

```
Object *object;
Posn3d p;

/* scale object by factor of 2 about its midpoint. */
object_getposition(object, p);
object_scale(object, 2.0, p);
```

object_stretch

```
void object_stretch(object, factors, point, frame)
Object *object;
Posn3d factors;
Posn3d point;
short frame;
```

object_stretch stretches an object by applying a different scale factor in each of the 3 coordinate dimensions. This can be compared to the function object_scale which scales the object uniformly by applying the same scale factor in each dimension.

The arguments to this function are as follows. factors contains the 3 scale factors (for X,Y and Z) by which object is to be stretched. point is the world coordinate point about which the object is stretched. Finally, frame is the coordinate frame in which the scale factors are applied - either FRAME_WORLD or FRAME_LOCAL.

Keep in mind that stretching an object changes its shape, while object_scale does not (it simply makes the object larger or smaller). However, the object's orientation (and X,Y, and Z axes) are not affected by object_stretch. Therefore, if you stretch an object, the definition of the object's orientation and axes may no longer coincide with the description given under Figures 2.1 and 2.2, until the object is stretched back to its original shape.

Also note that an object which is stretched in a frame other than FRAME_LOCAL will appear skewed. A bounding box created with the function object_boundingbox for this object is also skewed to fit closely about the object. However, the function object_intersect tests for intersections with the rectilinear (i.e. not skewed) bounding box which is the closest fitting such box in the object's local reference frame, and so may return incorrect results for skewed objects.

```
Object *object;
Posn3d p, factors;

/* get object's midpoint */
object_getposition(object, p);

/* set factors for stretch */
factors[X] = 2.0;
factors[Y] = factors[Z] = 1.0;

/* stretch object's longest dimension by factor of 2 about its midpoint. */
object_stretch(object, factors, p, FRAME_LOCAL);
```

Object tasks

An object's task gives the object functionality, or can be thought of as defining the object's behavior. The task function for each object is called **once each time through the simulation loop. Figure 1.1** shows where object task functions are called with respect to the other events in the simulation loop.

object_settask

```
void object_settask(object, task)
Object *object;
void (*task) ();
```

object_settask assigns a task function to an object. The task function is called for the object each frame of the simulation loop. For example, the object can be given a velocity by having the task call object_translate. An object's task can also be used to define the way the object interacts with other objects.

object_settask can be called at any time to assign a new task to an object, or to change the object's task. If object_settask is called more than once for the same object, the object will perform only the last task assigned to it.

In a simulation, the order in which events occur can be important. For this reason, it may be useful to keep in mind that in the simulation loop object task functions are called after the universe action function is called.

An object's task function takes an Object pointer as its only argument. The object passed in when the function is called is the object to which the task has been assigned by the object_settask function.

The following is an example of a task function in which mouse button presses cause an object to roll about its longest dimension.

```
/* a task function to roll an object when mouse buttons are pressed */
void roll_task(object)
    Object *object;
{
    short buttons;
    float w;

    buttons = sensor_getmiscdata(sensor);
    w = PI/9.0;          /* roll 20 degrees each tick */
```



```
    if ( buttons&MOUSE_LEFTBUTTON ) {
        object_rotate(object, X, w, FRAME_LOCAL);
    }
    else if ( buttons&MOUSE_RIGHTBUTTON ) {
        object_rotate(object, X, -w, FRAME_LOCAL);
    }
}
```

This task function assumes that a mouse sensor object has been created, for example with the call:

```
Sensor *sensor;
sensor = sensor_new(mouse_open, mouse_close, mouse_rawdata,
                    NULL);
```

Then, to assign this task to an object, say a barbecue rotisserie, you would make the following call:

```
Object *rotisserie;
object_settask(rotisserie, roll_task);
```

object_deletetask

```
void object_deletetask(object)
Object *object;
```

object_deletetask removes an object's task function. If the object had not previously been assigned a task this function has no effect.

Object Hierarchies

An object hierarchy is a group of objects which moves together as a whole but whose sub parts can move independently. As an example, we consider a hierarchically assembled robot arm such as is illustrated in Figure 2.3.

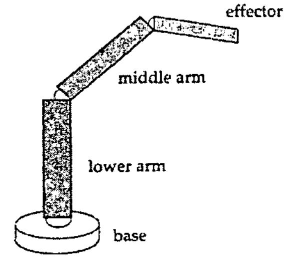


Figure 2.3. Hierarchically assembled robot arm.

Each part of the robot arm - the base, the lower segment, the middle segment, and the effector - must be created as a separate object, using the function object_new (or using object_copy once at least one of the arm objects was created). Let's say that pointers to these four objects are called base, lower, middle, and effector. Then to assemble the robot arm as in Figure 2.3, you would make the following calls to object_attach:

```
object_attach(base, lower);
object_attach(lower, middle);
object_attach(middle, effector);
```

These calls result in an object hierarchy in which base is the root, and moving down through the hierarchy we find lower, then middle, and then effector. (Don't be confused by the fact that "down" in the hierarchy corresponds to "up" in Figure 2.3!) When an object in the

hierarchy moves, it moves all of the objects that are hierarchically below it, as if the objects were rigidly attached. Objects that are hierarchically above the object are not affected by the object's motion. For example, when the lower arm moves, this causes the middle arm and effector to move with it, while the base is unaffected. When the effector moves, none of the other objects are affected because the effector is at the bottom of the hierarchy. Since sub-objects move automatically with their parent objects, if you wish to move an entire object hierarchy, you need only call the move function for the topmost object in the hierarchy. In the robot arm example, to move the entire arm you would simply move the base. The functions object_scale and object_stretch, however, are applied only to the object they are called for.

object_setpivot is a useful function when creating object hierarchies. An object's pivot point is the point that the object rotates about (unless object_rotatepoint is called). By default, an object's pivot point is its midpoint. However, in the robot arm model, it would be undesirable to have the arm segments rotate about their midpoints, because they would become detached from the neighboring arm segments. Instead, the pivot points should be located at the ball joints, in other words, where each arm segment meets the one that is above it in the hierarchy.

The following code fragment shows how the pivot point for the robot arm effector might be determined. It uses the object's local reference frame (the fact that the X axis typically points along an object's longest dimension) and extents to do this.

```
Posn3d xaxis;      /* points along longest dimension of effector */
Posn3d extents;    /* we'll use extents[X], the longest dimension */
Posn3d midpt;      /* midpoint of effector arm */
Posn3d pivot;      /* new pivot point */
Object *effector;

/* get the effector's midpoint, extents,
and unit vector in x direction in the effector's reference frame. */
object_getposition(effector, midpt);
object_getextents(effector, extents);
object_getaxis(effector, X, xaxis);
```

```
/* multiply half of extents[X] in to unit vector along x axis of effector
to get vector between effector midpoint and desired pivot point */
mult_sv(0.5*extents[X], xaxis);
subtract(midpt, xaxis, pivot);

/* set the pivot point */
object_setpivot(effector, pivot);
```

Any type of sensor could be attached to the various robot arm segments to cause the arm to move. Most likely, only rotation input from these sensors would be applied to the robot arm segments so that each segment would simply rotate about its pivot point and not become detached from the arm segment hierarchically above it. For example, if using a Spaceball or Geometry Ball Jr. to control part of the robot arm, you could constrain the device to return only rotations with the call:

```
sensor_setconstraints(sensor, XCON | YCON | ZCON);
```

Keyboard input, mouse button presses, or other device input could also be used to control the robot arm. For example, let us assume that in your main program you have established the pivot point of the effector. Then, to rotate the effector using the left mouse button, you could use the following effector task function (assigned with object_settask). Note that the following code fragment assumes the existence of a global sensor handle called mouse to a mouse sensor object.

```
void effector_task(o)
    Object *o;
{
    float w;       /* amount of rotation (radians) */

    /* return if the left mouse button isn't pressed. */
    if ( ! (sensor_getmiscdata(mouse) & MOUSE_LEFTBUTTON) )
        return;

    /* Rotate the object about its pivot point.
```



```
    Rotation about the effector's Y or Z axis will cause the
    arm to pitch or yaw, rather than to twist about its length. */
    w = sensor_getangularrate(mouse);
    object_rotate(o, Y, w, FRAME_LOCAL);
}
```

If you wish to delete an object that is part of a hierarchy, or if you wish to remove it from the simulation, you must first disassemble it from the hierarchy with the appropriate object_detach calls. Similarly, if you wish to insert an object into the middle of a hierarchy, you must first disassemble the hierarchy at the insertion point, and then re-assemble it with the new object.

object_attach

```
void object_attach(object1, object2)
Object *object1, *object2;
```

object_attach attaches object2 hierarchically below object1.

See also object_setpivot, which may be useful for defining the point about which the sub object rotates.

object_detach

```
FLAG object_detach(object1, object2)
Object *object1, *object2;
```

If object2 is a sub-object of object1, then object_detach removes object2 from below object1. If object2 is not a sub-object of object1, then this call has no effect.

Sensor Management

object_addsensor

```
void object_addsensor(object, sensor, frame)
Object *object;
Sensor *sensor;
short frame;
```

object_addsensor attaches a sensor to an object. Until the sensor is removed from the object (with a call to object_removesensor), input from the sensor will result in motion of the object. The sensor will cause the object to move in the reference frame passed in to object_addsensor. This reference frame is either FRAME_WORLD, FRAME_LOCAL, or FRAME_VPOINT.

When using FRAME_LOCAL it may be necessary to initially align the reference frame of the sensor with that of the object. The function sensor_rotate is used for this purpose.

For a given object, the reference frame in which the object is affected by sensors is can be set with object_setsensorframe or with object_addsensor (as described above). The reference frame in which sensors affect the object's motion will be the last one that was set with either of these calls.

object_removesensor

```
void object_removesensor(object, sensor)
Object *object;
Sensor *sensor;
```

object_removesensor detaches a sensor from an object so that input from the sensor no longer causes the object to move.

object_setsensorframe

```
void object_setsensorframe(object, frame)
Object *object;
short frame;
```

object_setsensorframe sets the reference frame in which sensors attached to the object move the object. The reference frame is FRAME_WORLD, FRAME_LOCAL, or FRAME_VPOINT.

object_getsensorframe

```
short object_getsensorframe(object)
Object *object;
```

object_getsensorframe returns the reference frame in which sensors attached to the object affect the object's motion.

For a given object, the reference frame in which the object is affected by sensors can be set with object_setsensorframe or with object_addsensor (as described above). The reference frame in which sensors affect the object's motion is the last one set with either of these calls.

Object texturing

The following two functions deal with the application and removal of bitmap textures to each surface of an object. More information on the use of textures in WorldToolKit is given in Chapter 9.

object_texture_apply

```
FLAG object_texture_apply(object, filename, shaded, transparent)
Object *object;
char *filename;
FLAG shaded;
FLAG transparent;
```

object_texture_apply applies a texture to each surface of an object.

The texture that is applied is the bitmap stored in a file with name filename. If any of the polygons in the object already had a texture applied, the old texture is replaced by the new texture. The FLAG arguments passed in indicate whether the texture is to be shaded or transparent. Presently, textures cannot be both shaded and transparent. **If both flags are given as true, transparency will be used.**

The texture bitmap must be in DVI ".i16" format as described in Chapter 9, at 128x120 resolution. It is searched for in the current directory, and along the path given by the VIM environment variable in DOS.

The function returns TRUE if the texture could be applied. In situations where the texture could not be found, as when the file named filename is not found, the function returns FALSE.

The way in which the texture is applied to the object surfaces is described under the function poly_texture_apply in Chapter 9.

See also object_texture_delete and poly_texture_delete.

object_texture_delete

```
void object_texture_delete(object)
Object *object;
```

object_texture_delete removes all textures from an object's surfaces, regardless of the way in which the textures were applied.

See also object_texture_apply, poly_texture_apply, and poly_texture_delete.

Terrain objects

Terrain objects are graphical objects representing landscape. These objects are special in that they can be defined and created within WorldToolKit rather than originating in another modelling program.

Functions for creating three types of terrain objects are provided:

1. terrain_flat creates a flat checkerboard terrain object. This kind of terrain can be helpful for visualizing perspective.
2. terrain_random creates a terrain object with random altitude values. This function is useful for creating an interesting terrain landscape with a minimum of effort.
3. terrain_data creates a terrain object whose altitude values are read in from a file. This lets you create terrain from real-world data or from any data you wish.

Each of these is an object constructor similar to object_new, except they create the geometric data implicitly rather than reading it from a file. As with other constructors, they do not take an object handle as their first argument, but instead return a new object they have created. As with object_new, the graphical terrain object constructed by the above functions is automatically added to the universe, so that the function object_add does not have to be called.

As with calls to object_new, calls to any of the above terrain functions must occur after the function universe_load is called (if universe_load is called by your program). This is because universe_load automatically calls universe_vacuum to make way for new graphics being loaded in. In addition, after any calls to the terrain functions, and before calling universe_go, you must call universe_ready to establish the colortable.

terrain_flat

```
Object *terrain_flat(altitude, nx, nz, x0, z0, lengthx, lengthz, c1, c2)
float altitude;
short nx, nz;
float x0, z0;
float lengthx, lengthz;
short c1, c2;
```

terrain_flat returns a checkerboard-like graphical object parallel to the X-Z plane of the world coordinate frame. The terrain is located at the altitude (that is, Y value) passed in. The rectangular polygons making up the checkerboard are colored alternately with colors c1 and c2, whose values are shorts in the range [0x000, 0xfff]. In other words, c1 and c2 are represented with 4 bits for each of R, G, and B.

The checkerboard is a rectangle with x0 and z0 the minimum X and Z values. The checkerboard is an array of nx by nz rectangular polygons whose sides are lengthx and lengthz in the X and Z dimensions respectively.

Keep in mind that the Y axis in the world coordinate frame points downward, so that as altitude increases, the level of the terrain actually decreases.

terrain_random

```
Object *terrain_random(altitude, mag, nsteps, nx, nz, x0, z0,
                       lengthx, lengthz, c)
float altitude;
float mag;
short nsteps;
short nx, nz;
float x0, z0;
float lengthx, lengthz;
short c;
```

terrain_random returns a graphical terrain object with random altitude values.

The terrain consists of a regular grid of terrain vertices in the X-Z plane, together with a random altitude value at each vertex. Each random altitude value is between altitude + mag and altitude - mag. The number of different altitude values (i.e. random numbers) that are generated is nsteps or less.

The terrain grid has nx by nz terrain patches whose sides are of lengthx and lengthz in the X and Z dimensions respectively. The grid is a rectangular array whose minimum x and z values are x0 and z0. Each terrain patch has color c, whose value is a short in the range [0x000, 0xfff]. In other words, c is represented with 4 bits for each of R, G, and B.

Object *x ;

x = terrain_random (.., ..., ...);

terrain_data

```
Object *terrain_data(filename, nx, nz, x0, z0, lengthx, lengthz, c)
char *filename;
short nx, nz;
float x0, z0;
float lengthx, lengthz;
short c;
```

terrain_data returns a graphical terrain object whose altitude values are specified in the file named filename.

The terrain grid has nx by nz terrain patches whose sides are of lengthx and lengthz in the X and Z dimensions respectively. The grid is a rectangular array whose minimum X and Z values are x0 and z0. Each terrain patch has color c, whose value is a short in the range [0x000, 0xfff]. In other words, c is represented with 4 bits for each of R, G, and B.

The format of the terrain data file is as follows. Since the number of terrain patches is nx times nz, the number of terrain vertices is (nx + 1) times (nz + 1). The terrain data file must contain one floating point value for each vertex. These values are specified as (nz + 1) lines of data, with each line containing (nx + 1) floating point values separated by spaces. Reading across a given line corresponds to increasing x values on the terrain grid, while reading down a column corresponds to increasing z values. Lines are terminated with a carriage return, and there should be no additional punctuation.

The following is a simple example of a terrain file appropriate for the creation of an array of terrain patches with nx = 2 and nz = 3.

```
200.0 250.0 200.0
200.0 300.0 230.0
250.0 224.0 300.0
200.0 250.0 200.0
```


User-specifiable object data

A void * pointer is included as part of the structure defining an object, so that you can store whatever data you wish with a graphical object. The following functions can be used to set and get this field within any object.

object_setdata

```
void object_setdata(object, data)
Object *object;
void *data;
```

To set the user-defined data field in an object, use this function. Private application data can be stored in any struct. Pass a pointer to such a struct, cast to a void*, in as the data argument to store a pointer to the struct within the object. See the example under object_getdata.

Note that private object data is not stored with the object by call to object_save; storing private data is the responsibility of the application program.

object_getdata

```
void *object_getdata(object)
Object *object;
```

This function retrieves private data stored within an object. You should cast the value returned by this function to the same type used to store the data in the object using the object_setdata function.

For example, let us say that you wish to store a file pointer within an object. The following section of code illustrates this:

```
FILE *myfp;
Object *object;    /* object where we will stash the file pointer */

/* place myfp within the object. Note mandatory cast to void* */
object_setdata(object, (void*)myfp);

myfp = NULL;    /* destroy it */

/* retrieve the pointer from within the object.
Note the cast need to retrieve the type.*/
myfp = (FILE*) object_getdata(object);
```

Performance / Statistics functions

object_npolygons

```
long object_npolygons(object)
Object *object;
```

This function returns the total number of polygons present in the object. See also universe_npolygons, for a count of the number of polygons in all objects in the universe.

3 Sensors

Introduction to sensor objects

Sensor objects in WorldToolKit generate position, orientation, and other kinds of data by reading inputs which originate in the real world. These inputs can be used to control motion and other behavioral aspects of objects in the simulation. Sensors permit the user of a WorldToolKit application to be directly coupled to the viewpoint or objects in the universe.

Many of the 3D and 6D (position/orientation) sensors available on the market are explicitly supported by WorldToolKit. There are two principal classes of such sensors: desk-based sensors and sensors that are worn on various parts of the body. While most desk-based sensors generate relative inputs, that is, *changes* in position and orientation, devices worn on the body typically generate absolute records, that is, values that correspond to their specific spatial location.

In the former category are conventional devices, such as the mouse, and isometric balls such as the CiS Geometry Ball Jr. and Spaceball Technology's Spaceball that respond to forces and torques applied by the user. Using such devices, a 3D object can be directly manipulated, displaced or rotated, with the ball acting as if directly connected to the object. Ball sensors are also useful for moving the viewpoint, with applied displacements and rotational forces acting to move and rotate the viewpoint. In this mode of operation, with a ball sensor attached to the viewpoint, the ball operates like a "fly-by-wire" helicopter.

The second category of sensor (sensors generating absolute records) includes electromagnetic 6D trackers such as the Polhemus 3Space and Ascension Bird. This type of sensor can be used for viewpoint tracking when affixed to a head-mounted display. In addition to electromagnetic devices, a variety of ultrasonic ranging/triangulation devices and optical devices exist for absolute position and orientation tracking.

Regardless of the underlying hardware technology by which they operate, WorldToolKit's sensor objects are treated homogeneously and can be used interchangeably in an application. Once a sensor object is created, it is automatically maintained by the simulation manager, as are the objects to which the sensor is attached. In this way, the developer does not have to deal directly with considerations such as whether the sensor is returning relative or absolute records, or whether it is polled or streaming its data.

WorldToolKit provides drivers for the following devices, making them easy to connect to your computer and use in your applications:

1. Mouse (Microsoft, Logitech, or compatible)
2. Spaceball Technology Spaceball
3. CIS Graphics Geometry Ball Jr.
4. Polhemus 3Space Isotrak
5. Ascension Bird

To use a device which is not currently supported, consult Appendix D "Writing a Sensor Driver".

Sensor lag and frame-rate

WorldToolKit has been designed specifically so that users can interact with computer-generated graphics flexibly and in so-called "real-time". Sensor objects provide a means of accomplishing this by directly

coupling the user of an application to the geometry in the virtual world. The effectiveness of this interaction depends on several factors:

1. sensor lag (inversely related to sensor speed) - the time from when the sensor's state in the real world changes to when the sensor generates a record corresponding to that state.
2. sensor accuracy - the range of values that a sensor may return when in a given state. This is usually specified as something like: "+ or - 0.1 inches within a range of 8 feet".
3. frame-rate - the number of frames per second that the system displays.

WorldToolKit is intended to support applications which permit the user to be directly coupled to the geometry presented in the virtual world display. Even if your application runs with a high frame-rate, if the sensor lag is very large, then the user's impression of being able interact in the virtual world may suffer. For very precise manipulations within the virtual world, the shorter the lag time the better for the user to have adequate control.

Sensor object construction and destruction

Sensor objects can be created with either the generic sensor constructor function sensor_new or with one of WorldToolKit's device-specific constructor macros (geoball_new, spaceball_new, polhemus_new, or bird_new).

When should you use sensor_new and when should you use a device-specific constructor function? To answer this question, let's consider the case of creating a sensor object for a Geometry Ball Jr. connected to serial port COM1.

The device-specific constructor function for the Geometry Ball Jr. is a macro defined as follows:

```
#define geoball_new(port)
    sensor_new(geoball_open, geoball_close, geoball_update, \
        serial_new(port, \
                   (port==COM1? 0x7c:0x7b), \
                   (BAUD96 | NP | DATA8 | STOP1), \
                   12))
```

To use this macro, you would make the call:

```
Sensor *ball;
ball = geoball_new(COM1);
```

All of the device-specific constructors are simply macro calls to sensor_new, which takes as its first three arguments pointers to WorldToolKit functions which respectively open, close, and update the particular device.

So, the answer to the question is this: If the open, close, and update functions specified in the device-specific constructor function macro are appropriate for your application, you might as well just make the macro call. If you wish to use open, close, or update functions other than the default WorldToolKit driver functions, use sensor_new. The last argument to sensor_new is a serial port object, which typically you will not need to modify.

For example, if you wanted to use your own update function mygeoball_update for the Geometry Ball Jr. (consult Appendix D to find out what needs to be in such a function), you would create the sensor object with the call:

```
Sensor *ball;
ball = sensor_new(geoball_open, geoball_close, mygeoball_update,
                  serial_new(port, (port==COM1? 0x7c:0x7b),
                  (BAUD96 | NP | DATA8 | STOP1),
                  12));
```


You could also hard-code the use of either COM1 or COM2.

The next part of this chapter describes the functions which apply to sensor objects in general. Information about the specific devices supported in WorldToolKit is provided at the end of this chapter.

sensor_new

```
Sensor *sensor_new(openfn, closefn, updatefn, serial)
void (*openfn) ();
void (*closefn) ();
void (*updatefn) ();
Serial *serial;
```

sensor_new creates a new sensor object and adds it to the universe.

The first three arguments are pointers to functions to respectively initialize, terminate, and update a sensor. When we say that a particular device is supported in WorldToolKit, we mean that these functions are provided for that device. The names of the open, close, and update functions for devices supported in WorldToolKit are as follows. (Note that several update functions are provided for the mouse.)

1. **Mouse.** mouse_open; mouse_close; mouse_rawdata, mouse_drawcursor; mouse_moveview1; mouse_moveview2.
2. **Geometry Ball Jr.** geoball_open; geoball_close; geoball_update.
3. **Spaceball.** spaceball_open; spaceball_close; spaceball_update.
4. **Polhemus.** polhemus_open; polhemus_close; polhemus_update.
5. **Bird.** bird_open; bird_close; bird_update.

To use a device which is not yet supported, you must provide the open, close, and update functions. How to do so is described in Appendix D "Writing a sensor driver".

The openfn is called once when the sensor is created. The closefn is called once when the sensor is deleted, by call to sensor_delete (or

implicitly, by universe_delete). The updatefn is called by the WorldToolKit simulation manager once at the beginning of each frame. For sensors supported in WorldToolKit, openfn, closefn, and at least one updatefn have been written and are available in the library for use. For serial port devices, the argument serial is a pointer to an initialized serial port object. Otherwise serial is NULL.

sensor_delete

```
void sensor_delete(sensor)
    Sensor *sensor;
```

sensor_delete removes a sensor object from the universe, calls the sensor's closefn, and frees the memory occupied by the sensor.

sensor_next

```
Sensor *sensor_next(sensor)
    Sensor *sensor;
```

sensor_next returns the next sensor object in the list of sensors maintained by the universe. Call universe_getsensors to get the head of the sensor list. The sample code under the function sensor_setsensitivity illustrates the use of the sensor_next function.

Accessing sensor state

sensor_setsensitivity

```
void sensor_setsensitivity(sensor, s)
    Sensor *sensor;
    float s;
```

sensor_setsensitivity sets the sensitivity value of the sensor. A sensor's sensitivity value defines the maximum magnitude of the translational input from the sensor along each axis, in the same distance units as the 3D geometry making up the virtual world.

For example, suppose that you have a Spaceball attached to the universe's viewpoint. The Spaceball's sensitivity determines the maximum distance along each axis that your viewpoint will move when you push on the ball. To move faster, call sensor_setsensitivity with a larger value than is currently set for the device.

It will frequently be desirable to have the sensor's sensitivity scale with the size of the universe, or with some other characteristic distance scale in the virtual world. The example below shows how to accomplish this.

```
Sensor *sensor;
float radius;

/* request the universe radius */
radius = universe_getradius();

/* Iterate through all of the sensors in the universe,
   scaling sensor sensitivity with the size of the universe */
for ( sensor=universe_getsensors() ; sensor ;
            sensor=sensor_next(sensor) ) {
    sensor_setsensitivity(sensor, 0.01 * radius);
}
```

In this example, if the sensor is a Spaceball attached to your viewpoint, then each time through the simulation loop your viewpoint will move a distance equal to at most one hundredth of the universe's radius along each of X, Y, and Z. If you do not push on the Spaceball very hard, then you will move less than that.

Attempts to set a sensor's sensitivity to a negative value are rejected, with no change to the current sensitivity.

sensor_getsensitivity

```
float sensor_getsensitivity(sensor)
Sensor *sensor;
```

sensor_getsensitivity returns the sensor's sensitivity value. This value is described above under the function sensor_setsensitivity.

sensor_setangularrate

```
void sensor_setangularrate(sensor, s)
Sensor *sensor;
float s;
```

Several of the devices supported in WorldToolKit have built in to their update functions scaling of the rotation records received from the device. The devices which are scaled are the Spaceball, Geometry Ball Jr., and mouse (in the update functions mouse_moveview1 and mouse_moveview2). **You can not set the rotational speed of the Polhemus or Bird devices.**

sensor_setangularrate sets the scale factor for rotation records. The angular rate is the maximum rotation (in radians) around any axis that

a sensor will return in any pass through the simulation loop. It may be convenient to specify the angular rate in terms of the defined constant PI. For example:

```
Sensor *spaceball;

/* create the spaceball sensor object */
spaceball = spaceball_new(COM1);

/* set the maximum rotation from the spaceball around any axis
to 22.5 degrees per tick. */
sensor_setangularrate(spaceball, PI/8.0);

/* scale translational inputs with the size of the universe */
sensor_setsensitivity(spaceball, 0.01 * universe_getradius());
```

sensor_getangularrate

```
float sensor_getangularrate(sensor)
Sensor *sensor;
```

Several of the devices supported in WorldToolKit have scaling of the rotation records received from the device built in to their update functions . These devices are listed under sensor_setangularrate.

sensor_getangularrate returns the maximum angular rate of change around each axis for a given sensor. In other words, the maximum rotation about any axis in any pass through the simulation loop is at most this value.

Angular rate is specified in radians.

sensor_gettranslation

```
void sensor_gettranslation(sensor, translation)
Sensor *sensor;
Posn3d translation;
```

sensor_gettranslation retrieves the current translation record from the sensor and stores it in translation.

If the device is an absolute sensor such as the Polhemus, then translation is the change in sensor position since the last time through the simulation loop.

The translation record reflects any constraints that have been set for the device and the sensor's sensitivity scale factor. (See the functions sensor_setsensitivity and sensor_setconstraints.)

The following is an example of using a desktop device such as the Spaceball or Geometry Ball Jr. to interactively stretch an object. In this example, the sensor's translation record is obtained and then transformed so that the resulting scale factor for each coordinate lies between 0.0 and 2.0. (Values less than 1.0 make the object smaller, while values greater than 1.0 make it get larger.)

```
Object *object;
Sensor *ball;
Posn3d scalefactor;     /* for object_stretch */
Posn3d pos;             /* object midpoint */
float sensitivity;

sensor_gettranslation(ball, scalefactor);
sensitivity = sensor_getsensitivity(ball);

/* transform translation values to be between 0.0 and 2.0.
(each scalefactor[i] is between -sensitivity and +sensitivity.) */
for ( i=0 ; i<3 ; i++ ) {
    scalefactor[i] = 1.0 + scalefactor[i]/sensitivity;
}
```



```
/* stretch the object */
object_getposition(object, pos);
object_stretch(object, scalefactors, pos, FRAME_LOCAL);
```

sensor_getrotation

```
void sensor_getrotation(sensor, rotation)
Sensor *sensor;
Quat rotation;
```

sensor_getrotation retrieves the current rotation record from the sensor and stores it as a quaternion in rotation.

If the device is an absolute sensor such as the Polhemus, then rotation is the change in orientation since the last time through the simulation loop.

sensor_getmiscdata

```
short sensor_getmiscdata(sensor)
Sensor *sensor;
```

sensor_getmiscdata returns a short in which miscellaneous data pertaining to the sensor has been stored. For example, button press information is retrieved this way. Appendix A specifies the WorldToolKit defined constants which can be used to access this data.

For example, to detect a left-button press on the Geometry Ball Jr., you might have the following (which assumes that a Geometry Ball Jr. sensor object has been constructed and is pointed to by geoball):

```
if ( sensor_getmiscdata(geoball) & GEOBALL_LEFTBUTTON ) {
    printf("Left button press\n");
}
```

sensor_setrawdata

```
void sensor_setrawdata(sensor, dataptr)
Sensor *sensor;
void *dataptr;
```

To set the user-defined data field within a sensor object, use this function. dataptr must be explicitly cast to a void * when passed in to sensor_setrawdata, as in the following example which stores a Posn6d with a sensor:

```
Sensor *sensor;
Posn6d *mydata;
sensor_setrawdata(sensor, (void *)mydata);
```

sensor_getrawdata

```
void *sensor_getrawdata(sensor)
Sensor *sensor;
```

This function returns the sensor-specific raw data struct. This should be typecast into an appropriate value for a user or WorldToolKit-defined sensor. For example, for the mouse as implemented in WorldToolKit, the raw data is a Posn2d containing the current mouse cursor position in screen coordinates. This might be needed as an argument to a pick function, as in:

```
Sensor *mouse;
Poly *p;

/* create the mouse sensor object */
mouse = sensor_new(mouse_open, mouse_close,
                   mouse_moveview1, NULL);

/* pick polygon under the mouse cursor */
p = universe_pickpolygon(*(Posn2d*)sensor_getrawdata(mouse));
```


sensor_getserial

```
Serial *sensor_getserial(sensor)
Sensor *sensor;
```

This function returns the serial port object associated with a sensor. The serial port object is the same as that supplied as the final argument to the sensor_new call. It will be used chiefly by developers writing their own sensor drivers for devices not explicitly supported in WorldToolKit. See Appendix D for examples of its use.

Rotating sensor input

Each 6D sensor supported in WorldToolKit has a reference frame (that is, a set of coordinate axes) associated with it. This reference frame defines how input from the device generates X,Y, and Z translation and rotation sensor records. The reference frame convention for devices supported in WorldToolKit is described under the section for each device at the end of this chapter. As in the example below, these conventions have been chosen for their convenience when controlling a viewpoint or object in the reference frame of the viewpoint. In some cases however it may be necessary to use coordinate axes other than the default ones. For this reason the function sensor_rotate is provided.

Let's consider the Geometry Ball Jr. The coordinate axis convention for this device is that if it is sitting on your desk with the cord running out the back of the device away from you, then the Z axis points straight back (in the direction of the cord), the X axis points to the right, and the Y axis points straight down. (See Figure 3.1.) This coordinate convention was chosen for its convenience. Let's say that you are using the Geometry Ball Jr. to control your viewpoint, as set up with the following calls:

sensor_getconstraints

```
short sensor_getconstraints(sensor)
Sensor *sensor;
```

sensor_getconstraints returns a short describing the constraints currently imposed on the values returned by the sensor. To determine whether a particular constraint has been set, you can use the bitwise AND operator '&' for the particular constraint. For example:

```
Sensor *sensor;

if ( sensor_getconstraints(sensor) & XROTCON ) {
    printf("X rotations are constrained\n");
}
```

Functions for writing your own sensor driver

The following functions should only be used if you are writing your own sensor driver. Consult Appendix D for more on this subject.

sensor_setrecord

```
void sensor_setrecord(sensor, p, q)
Sensor *sensor;
Posn3d p;
Quat q;
```

Use this function to store the current relative position and orientation record with your sensor. If your sensor returns absolute records, you must call sensor_relativizerecord first.

You may also wish to apply constraints or scale factors to the sensor record before calling sensor_setrecord, as described in Appendix D.

sensor_relativizerecord

```
void sensor_relativizerecord(sensor, absolute_p, absolute_q, p, q)
Sensor *sensor;
Posn3d absolute_p, p;
Quat absolute_q, q;
```

If your sensor returns absolute records, use this function to generate the corresponding relative record. sensor_relativizerecord is passed the absolute position/orientation record obtained from your device this tick, and returns (in p and q) the change in position and orientation since last tick.

sensor_setlastrecord

```
void sensor_setlastrecord(sensor, absolute_p, absolute_q)
Sensor *sensor;
Posn3d absolute_p;
Quat absolute_q;
```

This function is needed only for sensors which return absolute position/orientation records. In your sensor update function, after you have set the new sensor record with sensor_setrecord, store the absolute record with sensor_setlastrecord so that the next record can be relativized with respect to it the next time through the simulation loop.

sensor_getlastrecord

```
void sensor_getlastrecord(sensor, absolute_p, absolute_q)
Sensor *sensor;
Posn3d absolute_p;
Quat absolute_q;
```

This function retrieves the position and orientation record previously set with the function sensor_setlastrecord and stores them in absolute_p and absolute_q.

sensor_setmiscdata

```
void sensor_setmiscdata(sensor, x)
Sensor *sensor;
short x;
```

Use sensor_setmiscdata to store miscellaneous sensor data with the sensor object. Typically, you will not need to use this function unless you are writing your own sensor driver. Appendix A lists the defined constants for button presses and other data for supported devices.

sensor_setupdatefn

```
void sensor_setupdatefn(sensor, updatefn)
Sensor *sensor;
void (*updatefn)();
```

sensor_setupdatefn lets you change a sensor's update function. A sensor object's update function is initially set in the sensor object constructor function object_new. sensor_setupdatefn should only be called if, after having created the sensor object, you wish to change its update function. The following example illustrates how to set a mouse

sensor's update function to the WorldToolKit supplied function mouse_moveview2. This example presumes that mouse was originally created as a pointer to a sensor object with open and close functions that properly initialize and close a mouse device (see the functions mouse_open and mouse_close).

```
Sensor *mouse;
sensor_setupdatefn(mouse, mouse_moveview2);
```

Mouse

WorldToolKit provides the following sensor driver functions for using the mouse: a function for opening the mouse device, a function for closing the mouse device, and several update functions. These functions are used only when calling sensor_new to create a new mouse sensor object, or when calling sensor_setupdatefn to change the mouse's update function.

When creating a mouse sensor object, you can use one of the update functions provided, or you may write your own. This is not a difficult thing to do. Your update function should first call mouse_rawdata to obtain the raw mouse record. It should then specify how the raw data is to be transformed into the 3D position and orientation record. Finally, your update function must store this record with the sensor by calling sensor_setrecord. An example mouse update function is given in Example 1 in Appendix D "Writing a sensor driver".

mouse_open

mouse_open opens the mouse device. Use this function only as an argument to sensor_new when creating a mouse sensor object.

mouse_close

mouse_close closes the mouse device. Use this function only as an argument to sensor_new when creating a mouse sensor object.

mouse_rawdata

mouse_rawdata is a mouse update function that simply reads in the x,y raw data from the mouse and stores it in the sensor's rawdata struct. This information is accessed using the function sensor_getrawdata as in the example below. Mouse button presses are also read, and can be accessed by calling sensor_getmiscdata.

The rawdata struct for the mouse is typedefed as:

```
typedef struct mouse_rawdata {
    float x,y;
} Mouse_rawdata;
```

The following is an example of accessing raw data read from the mouse. Note the use of defined constants MOUSE_LEFTBUTTON and MOUSE_RIGHTBUTTON. Another example of using mouse_rawdata is given in Example 1 of Appendix D.

```
void read_mouse_record(mouse)
    Sensor *mouse;
{
    Mouse_rawdata *pos;
    short rawbuttons;
    FLAG leftbutton, rightbutton, bothbuttons;

    /* get raw x and y mouse values in screen coordinates */
    pos = (Mouse_rawdata *)sensor_getrawdata(mouse);
    printf("Mouse position: %f, %f\n", pos->x, pos->y);
```



```
    /* get button press data */
    rawbuttons = sensor_getmiscdata(mouse);

    /* which buttons were pressed? */
    leftbutton = rawbuttons & MOUSE_LEFTBUTTON;
    rightbutton = rawbuttons & MOUSE_RIGHTBUTTON;
    bothbuttons = leftbutton && rightbutton;
}
```

mouse_moveview1

mouse_moveview1 is a mouse update function which can be useful for moving a viewpoint through a 3D environment. It is a fairly simple update function in that it does not pitch or roll the viewpoint. See mouse_moveview2 for a mouse update function providing control over all 6 degrees of freedom.

Maximum rotations and translations are scaled as described under the functions sensor_setangularrate and sensor_setsensitivity. In addition, translations and rotations can be constrained when using mouse_moveview1 with the function sensor_setconstraints.

With mouse_moveview1, a cursor is drawn on the screen tracking mouse movement. See the description under the update function mouse_drawcursor for more information about the cursor. Using mouse_moveview1, with the mouse attached to the viewpoint as in the example below, manipulating the mouse has the following effect:

1. When the mouse cursor is centered on the screen and no buttons are pressed, the viewpoint is stationary.
2. When the cursor is in the left half of the screen the viewpoint shifts to the left; in the right half of the screen the viewpoint shifts to the right.
3. When the cursor is in the top half of the screen the viewpoint moves forward; in the bottom half of the screen the viewpoint moves backward.
4. When the left mouse button is pressed the viewpoint yaws to the left;

when the right mouse button is pressed the viewpoint yaws to the right,

5. Pressing both mouse buttons simultaneously with the cursor in the top half of the screen translates the viewpoint upward. The viewpoint translates downward if the cursor is in the bottom half of the screen.

mouse_moveview1 also stuffs the rawdata struct Mouse_rawdata, so that the raw values can be accessed (as shown under the function mouse_rawdata) when mouse_moveview1 is specified as the update function.

The following is an example of creating a mouse sensor object with the mouse_moveview1 update function, and of attaching it to the viewpoint:

```
main()
{
    Sensor *mouse;

    /* initialize the universe */
    universe_new();

    /* Create the mouse sensor object.The last argument is NULL,
    since the mouse, although it may be connected to the serial port,
    is not treated by WorldToolKit as a serial port device */
    mouse = sensor_new(mouse_open, mouse_close,
                       mouse_moveview1, NULL);

    /* attach the mouse sensor to the viewpoint */
    viewpoint_addsensor(universe_getviewpoint(), mouse);

    /*........... */

    /* clean up */
    universe_delete();

    return 0;
}
```


mouse_moveview2

Like mouse_moveview1, mouse_moveview2 is a mouse update function which can be useful for moving a viewpoint through a 3D environment. Unlike mouse_moveview1 which translates and yaws the viewpoint, with mouse_moveview2 the viewpoint can also be pitched and rolled. Also unlike mouse_moveview1 which moves the viewpoint whenever the mouse cursor is away from the center of the screen, mouse_moveview2 only moves the viewpoint while the cursor is away from the center of the screen *and* one or more mouse buttons are pressed. The further away from the middle of the screen, the faster the movement. Maximum rotations and translations are scaled as described under the functions sensor_setangularrate and sensor_setsensitivity. In addition, translations and rotations can be constrained when using mouse_moveview2 with the function sensor_setconstraints.

Using mouse_moveview2, with the mouse attached to the viewpoint, manipulating the mouse has the following effect:

The viewpoint is stationary when neither the left nor the right mouse button is pressed.

The left mouse button lets you "walk" about your model. When the left button is pressed, and the cursor is:

1. in top half of screen - move forward
2. in bottom half of screen - move backward
3. in left half of screen - yaw left
4. in right half of screen - yaw right

When the right button is pressed, and the cursor is:

1. in top half of screen - move up
2. in bottom half of screen - move down
3. in left half of screen - pan left
4. in right half of screen - pan right

When the left AND right buttons are pressed, and the cursor is:

1. in top half of screen - pitch up
2. in bottom half of screen - pitch down
3. in left half of screen - roll left
4. in right half of screen - roll right

mouse_drawcursor

For a number of applications, such as interactively picking objects or polygons with the mouse, it is desirable to be able to see the position of the mouse. For this purpose, a mouse update function has been provided which tracks the mouse position and button presses as does mouse_rawdata, but in addition automatically draws a bitmap cursor at the current mouse position.

The drawing of the mouse cursor happens asynchronously, and its update rate is not affected by the frame rate of the universe being displayed. This makes it possible to effectively use the mouse for picking even in universes containing a very large number of polygons.

The bitmap used for the mouse cursor is found in the cursor.i16 file, which should be found on the path given by the VIM environment variable. Entirely black regions of the cursor are rendered transparently. Users may provide their own cursor bitmap by leaving it in the file named cursor.i16. The cursor size is currently hardcoded to 16x16 pixels.

To toggle the visibility of the cursor, sensor_setupdatefn can be called to replace the mouse_cursordraw update function with an alternate function which does not display the mouse cursor.

Spaceball and Geometry Ball Jr.

The Spaceball, from Spaceball Technology, Inc., and the Geometry Ball Jr., from CIS Graphics, Inc., are 6 degree of freedom serial port devices that sit on the desktop. They respond to both forces and torques, which can be mapped into translations and rotations in 3D.

The WorldToolKit update functions for the Spaceball and for the Geometry Ball Jr. package the translation and rotation record from the device into the sensor object's record, after:

1. transforming the record to the WorldToolKit coordinate convention,
2. applying any constraints which have been set with sensor_setconstraints, and,
3. applying any scale factors which may have been set with sensor_setsensitivity and sensor_setangularrate.

There are two ways to create a Spaceball or Geometry Ball Jr. sensor object in WorldToolKit. You can call sensor_new, passing in the driver functions (spaceball_open, spaceball_close, and spaceball_update, or geoball_open, geoball_close, and geoball_update) and a serial port object created for the Spaceball or Geometry Ball Jr. Or, these calls can be abbreviated through supplied macros to simply:

```
spaceball_new(COM1);
```

for example, to create a Spaceball object on serial port COM1, or:

```
geoball_new(COM1);
```

to create a Geometry Ball Jr. sensor object.

The coordinate frame of these sensors is defined in the WorldToolKit driver functions as follows. If the device is placed on a desk or table in front of you with the cable coming out the back of the device oriented away from you, then as illustrated in Figure 3.1, the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis

points down. If this coordinate frame is not appropriate for your application, the function sensor_rotate can be used to define the device's coordinate frame.

Polhemus and Bird

The Polhemus tracker from Polhemus Navigation Sciences, Inc., and the Bird from Ascension Technology Corporation are electromagnetic serial port devices that measure absolute position and orientation.

WorldToolKit provides the driver functions polhemus_open, polhemus_update, and polhemus_close for the Polhemus and bird_open, bird_update, and bird_close for the Bird. Just as for the Spaceball and Geometry Ball Jr., you can create a sensor object for a Polhemus attached to COM1 with the call:

```
polhemus_new(COM1);
```

or for the Bird attached to COM1 with the call:

```
bird_new(COM1);
```

The functions polhemus_new and bird_new assume that the dipswitches on the devices are configured as follows:

1. Polhemus: OFF ON OFF ON OFF
2. Bird: OFF ON OFF OFF OFF OFF OFF OFF

The above settings of the first three switches on each device signifies 9600 baud. To use other baud rates, create the sensor object with sensor_new rather than polhemus_new or bird_new, passing in a serial port object constructed for that baud rate.

When you call polhemus_new or bird_new to construct a new Polhemus or Bird sensor object, the openfn for the device is

automatically called. Part of the function of the openfn for these devices is to calibrate the sensor, which consists of obtaining an initial position and orientation record. This takes several seconds. Records subsequently generated by the updatefn are with respect to this initial reference frame. It may be useful in your application to let the user know that the device is about to be calibrated. For example, you might want to have a print statement such as:

```
Sensor *sensor;
printf("About to calibrate Polhemus...\n");
sensor = polhemus_new(COM1);
printf("Calibration complete.\n");
```

Translation records for Polhemus and Bird sensor objects can be scaled and/or constrained using the functions sensor_setsensitivity and sensor_setconstraints. It is useful, for example, to scale sensor inputs with the size of the universe. This can be accomplished as follows:

```
Sensor *bird;

/* create the bird sensor object */
bird = bird_new(COM1);

/* scale translations from the bird with the size of the universe */
sensor_setsensitivity(bird, 0.01 * universe_getradius());
```

Unlike translation records, however, orientation records from the Polhemus or Bird are not scaled in the WorldToolKit update functions for these devices. For example, if the Polhemus is used to track head motion (and the Polhemus sensor object is attached to the viewpoint), then a 360 degree rotation of the Polhemus device in the real world generates a 360 degree rotation in the virtual world. *In addition, orientation records from the Polhemus or Bird can not be constrained in the current version of WorldToolKit.*

The coordinate frame of these sensors is defined in the WorldToolKit driver functions as follows. If the receiver cube is placed "flat-end

down" in front of you with the cable from the cube coming out the back of the cube toward you, then as illustrated in Figure 3.2, the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function sensor_rotate can be used to define another coordinate frame for the device.

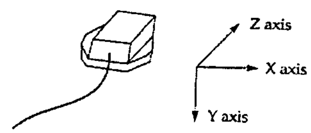


Figure 3.2. Polhemus and Bird sensor reference frame.

4 Lights

Introduction to light objects

WorldToolKit comes with two kinds of lighting: ambient light and directed light. Ambient light is background light used to illuminate all polygons equally regardless of their position or orientation. Directed lights have a specified position and orientation. The amount of brightness they contribute to the surface of a polygon depends on the dot product between the polygon normal and the direction of the light.

The intensity of the color of a polygon is determined by adding the contributions from each of the light sources in the universe. If the result is 0.0, then the polygon will be black, and if the result is 1.0, then the polygon will be of maximum brightness. Anything greater than 1.0 will also be considered to be maximum brightness. Therefore the ambient light level and intensity of each directed light should be between 0.0 and 1.0.

The shading of all graphical entities (stationary or moving) is recomputed whenever new lights are added, or when existing lights are removed or any of their parameters (position, direction, or intensity) are changed.

In addition, shading on an object's surfaces is automatically recomputed each frame that the object moves for any reason (e.g. because of attached sensor(s), tasks which affect the object, or through explicit calls to a function such as object_move).

Polygons do not cast shadows. Therefore, lighting on a polygon is not affected by polygons which might happen to be between it and a light source. Lights do not attenuate with distance from the polygon.

There is no limit on the number of lights that you may add to the simulation. However, the greater the number of lights, the greater the performance impact when the shading on an object's surfaces is recomputed. The time to compute the shading on an object's surfaces is proportional to the number of lights in the simulation. For this reason, if at any time you wish to turn a light "off", it is better to do so with a call to light_remove than by setting the light's intensity to 0.0 using light_setintensity. In the former case the light is removed from the simulation and no longer enters into shading computations; in the latter case the light remains part of the simulation.

Basic light management

light_new

```
Light *light_new(at, dir, intensity)
Posn3d at, dir;
float intensity;
```

light_new creates a new light and adds it to the universe's list of lights, returning a handle to the new light.

at is the position of the light, specified in world coordinates.

dir is the direction in which the light points, in world coordinates. The direction you pass in is normalized in light_new. However, if dir has zero magnitude, no light is constructed and light_new returns NULL.

The intensity of the light should be between 0.0 and 1.0.

light_delete

```
void light_delete(light)
Light *light;
```

light_delete removes a light from the universe and frees its memory.

Note that the function universe_delete deletes all of the universe's lights (by calling lights_deleteall), so that you do not have to explicitly delete lights at the end of your program if universe_delete is called, which it should be.

See also lights_deleteall.

light_add

```
void light_add(light)
Light *light;
```

light_add adds a light to the universe's list of lights. Note that when a light is constructed with light_new, it is automatically added to the universe. Use light_add when you wish to add a light to the universe which was previously removed from the universe with light_remove.

light_remove

```
void light_remove(light)
Light *light;
```

light_remove removes a light from the universe's list of lights.

light_remove is useful in applications in which lights are turned on and off. Turning a light "off" could be accomplished by setting its intensity to 0.0 (see light_setintensity). However, you will get better performance if you remove the light from the simulation with light_remove, because it then will no longer enter into shading computations.

lights_read

```
FLAG lights_read(filename)
char *filename;
```

lights_read reads in a list of light descriptions from an ASCII file and creates the corresponding light objects. Each light must be specified on a separate line of the file. For each light, seven floating point values must be specified: the X,Y, and Z coordinates of the light position, the X,Y, and Z coordinates of the light direction, and the light intensity.

Here is an example of creating three light from the file "lights_file":

```
lights_read("lights_file");
```

where lights_file has the form:

```
100.0 -180.0 -100.0 -0.58 0.58 0.58 0.6
-150.0 180.0 100.0 -0.58 0.58 -0.58 0.45
150.0 0.0 0.0 1.00 0.0 0.0 0.4
```

Notice that:

1. This file contains no punctuation other than a space between the floating point values.
2. Each direction vector (the 4th, 5th, and 6th values in each line) must truly have a direction. If you specify a direction vector whose length is not 1, lights_read will take care of normalizing it for you. However, if you pass in a direction vector whose magnitude is zero the light will not be constructed.
3. Intensity values must be between 0.0 and 1.0.

If the file passed in can not be opened, then lights_read returns FALSE. Otherwise lights_read returns TRUE.

lights_deleteall

```
void lights_deleteall();
```

lights_deleteall deletes all lights which have been created with light_new or lights_read. Ambient light is unaffected.

light_next

```
Light *light_next(light);
Light *light;
```

light_next returns the next light in the universe's list of light objects.

In this example, the lights in the universe are dimmed by reducing their intensity by a factor of 0.9.

```
Light *light;
float intensity;

/* iterate through the universe's lights, dimming them */
for ( light=universe_getlights() ; light ; light=light_next(light) ) {
    intensity = 0.9 * light_getintensity(light);
    light_setintensity(light, intensity);
}
```

Sensor management

Lights can have sensors attached to them, just as viewpoints and graphical objects can. When you attach a sensor to a light, input from the sensor device causes the light to move or be redirected. Shading on objects is automatically recomputed when this occurs.

light_addsensor

```
void light_addsensor(light, sensor, frame)
Light *light;
Sensor *sensor;
short frame;
```

light_addsensor attaches a sensor object to a light object, so that input from the sensor device causes the light to move.

sensor is a pointer to a sensor object, for example, as returned by sensor_new or one of the device-specific calls such as spaceball_new.

frame is the reference frame in which input from the sensor moves the light. This reference frame must be either FRAME_WORLD or FRAME_VPOINT (see Figure 2.2). If the frame is given as FRAME_LOCAL, the function returns with no effect.

It may be useful to have a graphical representation of a light source. To accomplish this, create a graphical object, move it to the same position as the light, and attach the same sensor to both the 3D object and the light (operating in the same reference frame, of course). The following is an example of a function which attaches a graphical object to a light:

```
void attach_object_to_light(light, object, sensor)
    Light *light;
    Object *object;
    Sensor *sensor;
{
    Posn3d lightpos;    /* light's position */

    /* move the object to where the light is */
    light_getposition(light, lightpos);
    object_setposition(object, lightpos);

    /* make sure that the object will rotate about the light's position */
    object_setpivot(object,lightpos);

    /* attach sensor to both light and object */
    light_addsensor(light, sensor, FRAME_WORLD);
    object_addsensor(object, sensor, FRAME_WORLD);
}
```


light_removesensor

```
void light_removesensor(light, sensor)
Light *light;
Sensor *sensor;
```

light_removesensor removes a sensor object from a light object, so that input from the sensor device no longer causes the light to move.

Ambient light

Ambient light illuminates the surfaces of graphical objects regardless of their position or orientation. Ambient light is always present. The default ambient light level is 0.4. The intensity of the ambient light can be retrieved and set with the following functions.

light_setambient

```
void light_setambient(x)
float x;
```

light_setambient sets the ambient light level to x. This value must be between 0.0 and 1.0.

light_getambient

```
float light_getambient();
```

light_getambient returns the value of the ambient light. In the following example the ambient light intensity is increased by 10%.

```
light_setambient(1.1 * light_getambient());
```

Accessing light properties

Light properties (position, direction, and intensity) can be accessed and changed. When changes are made to lights, the shading on graphical entities is automatically updated.

light_setposition

```
void light_setposition(light, p)
Light *light;
Posn3d p;
```

light_setposition changes the 3D position of a light.

An example of using this function is given under the function light_getdirection.

light_getposition

```
void light_getposition(light, p)
Light *light;
Posn3d p;
```

light_getposition stores the position of the light in p. Use light_setposition to change a light's position. An example of using this function is given under the function light_getdirection.

light_setdirection

```
vc'd light_setdirection(light, dir)
Light *light;
Posn3d dir;
```

light_setdirection sets the direction of a light to the vector passed in.

The direction vector passed in does not have to be normalized, that is, it is not required to have length equal to 1. (The function will automatically normalize the direction vector for you.) However, if the three components of the direction vector are all 0, the function returns without setting the light's direction. In the following example, a light's direction is set to lie in the X-Z plane, with a specified angle theta from the X axis.

```
void orientlight(light, theta)
    Light *light;
    double theta;
{
    Posn3d dir;
    dir[Y] = 0.0;
    dir[X] = cos(theta);
    dir[Z] = sin(theta);
    light_setdirection(light, dir);
}
```

of the actual colors present in the models. Intensity ramps are then constructed from these representatives.

The call to universe_ready, which must be made prior to universe_go and after any calls to universe_load, object_new, animation_new, or the terrain-creation functions, calculates optimal colortable entries, and updates indices to this table for all polygons in the universe. WorldToolKit users need not be aware of most of the details of this process, which is managed entirely by the WorldToolKit library.

If you wish to introduce new objects while the simulation is running, it is best to do so by calling object_new (for example) and then object_remove to remove it from the simulation, both outside of the simulation loop. This avoids the overhead of optimizing the colortable in the real-time loop. Then, when you call universe_ready, the colors for all objects (those in the simulation and those that have been removed) are determined. Then call universe_go to start the simulation. Finally, when you want the new object to appear, call object_add to add it to the simulation.

6 Viewpoints

Introduction to viewpoint objects

A viewpoint object contains parameters which define how the universe is is projected to the computer screen and rendered. At any given time, the universe has exactly one viewpoint object, and it is from this viewpoint that the universe is drawn. The viewpoint parameters for monoscopic and stereo viewing are illustrated in Figure 6.1 and are defined as follows:

1. **position** in the 3D virtual world. If the viewpoint is stereo, then the viewpoint position corresponds to that of the left eye view.
2. **orientation** in the 3D virtual world.
3. **direction**. The viewpoint's direction is a unit vector. The viewpoint's orientation can be thought of as being made up of the viewpoint's direction vector plus a twist about this vector.
4. **angle** (in radians). Think of the window or screen in which the virtual world is drawn as literally a window through which the world is viewed. Imagine that the scene is drawn as viewed by an eye centered in front of the window. Draw a line from the eye to the middle of the window, and another line from the eye to the right edge of the window. The view angle is the angle between these two lines.
5. **hither clipping plane value**. The distance (along the viewpoint direction) from the viewpoint position to the hither clipping plane. Graphical entities are clipped at this plane; only those portions of graphical entities on the opposite side of the hither plane from the viewpoint are drawn.
6. **stereo**. Is the view to be drawn as a stereo pair of images suitable for viewing from the left and right eye positions?

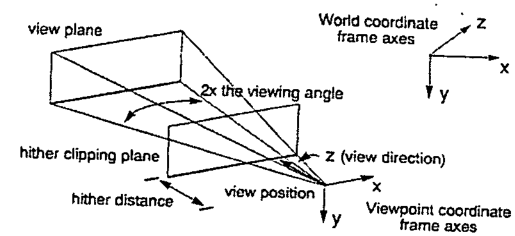
7. **parallax** (for stereo viewing only). Distance between the positions from which right and left eye views are drawn in the 3D virtual world, in the distance units for that world.
8. **convergence** (for stereo viewing only). The offset in pixels along the horizontal screen axis between the left and right eye images (for example, between the left edge of the right and left eye images).

Each of these parameters can be accessed or changed as described in the calls below. The default values for these parameters, assigned when a new viewpoint is constructed, are specified in the description of the call viewpoint_new. Note that a viewpoint's direction is not set directly, but is set when the viewpoint's orientation is set.

In addition, a viewpoint's position and orientation can be controlled by sensors attached to it. For example, let us say that a mouse sensor object is constructed and attached to the universe's viewpoint. Then as the mouse is moved, the viewpoint will move automatically.

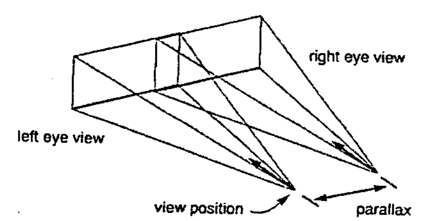
When the universe is created with universe_new, a viewpoint object is automatically created for it. For many applications, this one viewpoint will be sufficient. The universe's viewpoint is accessed with the call universe_getviewpoint.

For other applications, it may be convenient to construct additional viewpoints and to be able to switch between them. The function universe_setviewpoint is used to set the viewpoint from which the universe is drawn. An analogy to changing viewpoints in this way is cutting between various cameras in a movie.



The view position is the origin of the viewpoint coordinate frame. The view direction is the same as the Z axis in the viewpoint frame. In this illustration, the Y axes in the viewpoint frame and the world coordinate frame happen to be parallel, but this will not in general be the case.

6.1(a) Monoscopic viewing geometry.



Stereoscopic viewing has the same parameters as monoscopic viewing, except that there are 2 view pyramids (instead of one), linearly offset by the parallax distance. Stereo convergence is described in the text.

6.1(b) Stereoscopic viewing

Basic viewpoint management

viewpoint_new

```
Viewpoint *viewpoint_new();
```

viewpoint_new returns a new viewpoint object with the following default parameter values:

1. **position.** The origin of the world coordinate frame: (0.0, 0.0, 0.0)
2. **orientation.** Looking straight down the Z axis, with no twist about this axis. The corresponding quaternion is (0.0, 0.0, 0.0, 1.0), and the corresponding orientation matrix is the identity matrix. From this orientation, the world X axis points to the right, the world Y axis points straight down, and the world Z axis points dead ahead.
3. **direction.** Looking straight down the Z axis: (0.0, 0.0, 1.0)
4. **angle.** DEFAULT_VIEWANGLE = 0.698131 radians (40 degrees). As defined in the introduction to this chapter, this is actually half of the horizontal viewing angle.
5. **hither clipping plane value.** DEFAULT_HITHERVALUE = 0.05. The hither clipping plane is immediately in front of the viewpoint.
6. **stereo.** FALSE
7. **parallax.** 0.0. Both right and left eye views are from the same position.
8. **convergence.** 0.

Consult the introduction to this chapter and Figure 6.1 for definitions of these parameters.

The universe maintains a list of all viewpoints created with viewpoint_new, so that they can all be automatically deleted when universe_delete is called. However, unlike other object types, at any given time the universe has only one active viewpoint object. This is the viewpoint from which the universe is rendered. Use universe_setviewpoint to change the viewpoint used to render the universe.

If only one viewpoint is required for your application, you will not need to call viewpoint_new. This is because when you call universe_new, a viewpoint is automatically created and added to the universe, and becomes the universe's active viewpoint.

viewpoint_delete

```
void viewpoint_delete(view)
Viewpoint *view;
```

viewpoint_delete deletes a viewpoint object, freeing the memory it occupies. Since the universe *must* have a viewpoint from which it is drawn, calling viewpoint_delete for the universe's current viewpoint will have no effect.

viewpoint_copy

```
Viewpoint *viewpoint_copy(old_view);
Viewpoint *old_view;
```

This function copies an existing viewpoint, returning a handle to the copy. All the state of the old viewpoint is copied, except for any sensors which may have been attached to the old viewpoint. That is, the copied viewpoint does not have any sensors attached.

Sensor attachment and detachment

viewpoint_addsensor

```
void viewpoint_addsensor(view, sensor)
Viewpoint *view;
Sensor *sensor;
```

viewpoint_addsensor attaches a sensor to a viewpoint. Once a sensor is attached to the viewpoint, translation and orientation sensor records automatically cause the corresponding translation and rotation of the viewpoint. If a viewpoint has more than one sensor attached to it, each one contributes to motion of the viewpoint.

Sensors affect only the universe's current viewpoint. If a sensor is attached to a viewpoint which is not the current viewpoint, input from the sensor does not affect that viewpoint's position or orientation. If the viewpoint becomes the universe's current viewpoint (through a call to universe_setviewpoint), from that time sensor input does affect the viewpoint position and orientation.

In the following example, Polhemus and Spaceball sensor objects are created and attached to the viewpoint. This is a useful sensor configuration in setups in which head tracking with an absolute sensor such as the Polhemus is desired, but it is also desired to be able to independently control the viewpoint with a joystick-like device (such as the Spaceball).


```
#include <stdio.h>
#include "wt.h"

main()
{
    Sensor *polhemus, *spaceball;/* sensor objects */
    Posn6d x;            /* viewpoint from model */

    /* initialize the universe */
    universe_new();

    /* create some graphics */
    universe_load("myworld", &x, 1.0);

    /* create a polhemus sensor object on serial port COM1 */
    polhemus = polhemus_new(COM1);

    /* create a spaceball sensor object on serial port COM2 */
    spaceball = spaceball_new(COM2);

    /* attach the polhemus and spaceball to the universe's viewpoint */
    viewpoint_addsensor(universe_getviewpoint(), polhemus);
    viewpoint_addsensor(universe_getviewpoint(); spaceball);

    /* prepare to enter the simulation */
    universe_ready();

    /* start the simulation */
    universe_go();

    /* clean up */
    universe_delete();

    return 0;
}
```

The previous example uses an absolute device and a relative device to control the viewpoint, and is a fairly intuitive configuration to work with. It can be interesting to experiment with different sensor configurations. Not all work as well as others, although what works well depends on the particular application. Attaching more than one absolute sensor to the same viewpoint, for example, can lead to non-intuitive results if they generate input simultaneously.

viewpoint_removesensor

```
void viewpoint_removesensor(view, sensor)
Viewpoint *view;
Sensor *sensor;
```

viewpoint_removesensor detaches a sensor from a viewpoint object, so that input from the sensor no longer affects motion of the viewpoint.

Accessing viewpoint properties

viewpoint_setposition

```
void viewpoint_setposition(view, p)
Viewpoint *view;
Posn3d p;
```

viewpoint_setposition moves the viewpoint to the 3D position p passed in.

An example of using this function to move a viewpoint straight ahead is given under the function viewpoint_getdirection.

viewpoint_getposition

```
void viewpoint_getposition(view, p)
Viewpoint *view;
Posn3d p;
```

viewpoint_getposition sets p to the 3D position of the viewpoint.

An example of using this function to move a viewpoint straight ahead is given under the function viewpoint_getdirection.

viewpoint_setorientation

```
void viewpoint_setorientation(view, q)
Viewpoint *view;
Quat q;
```

viewpoint_setorientation sets the orientation of the viewpoint to q, where q is represented in quaternion form. If in your program orientations are represented as 3x3 matrices, the conversion function matrix_2_quat can be used to generate the corresponding quaternion which can then be passed in to viewpoint_setorientation.

viewpoint_getorientation

```
void viewpoint_getorientation(view, q)
Viewpoint *view;
Quat q;
```

viewpoint_getorientation sets q to the orientation of the viewpoint, specified as a quaternion. To convert this to a 3x3 matrix representation, you can use the function quat_2_matrix.

viewpoint_move

```
void viewpoint_move(view, newviewat)
Viewpoint *view;
Posn6d *newviewat;
```

viewpoint_move moves a viewpoint to a given position and orientation newviewat in one call. This function takes a *pointer* to a Posn6d struct (which contains both a Posn3d and a Quat), and moves the viewpoint to the absolute position and orientation contained in them.

Note that since a Posn6d is a struct (see Chapter 11: Math Library), only a pointer to it can be passed in to a function. Structs should not be directly passed in to functions.

When sensors are attached to a viewpoint, the viewpoint moves automatically with input from the device(s). Any calls made to viewpoint_move result in additional movement of the viewpoint.

viewpoint_getdirection

```
void viewpoint_getdirection(view, dir)
Viewpoint *view;
Posn3d dir;
```

viewpoint_getdirection sets dir to a unit vector in the direction of the viewpoint. Note that there is no corresponding viewpoint_setdirection function. This is because the viewpoint's direction is derived from its orientation and can not be set independently.

The following code fragment moves a viewpoint to a position 10 units ahead along the current viewing direction.

```
Viewpoint *view;
Posn3d dir, pos, newpos;

/* get the position and direction of the viewpoint */
viewpoint_getposition(view, pos);
viewpoint_getdirection(view, dir);

/* multiply the direction vector by 10.0 */
mult_sv(10.0, dir);

/* add the vector dir (which is now 10.0 units long) to
pos to get the new viewpoint position. */
add(pos, dir, newpos);

/* move the viewpoint to the new position */
viewpoint_setposition(view, newpos);
```

viewpoint_setviewangle

```
void viewpoint_setviewangle(view, angle)
Viewpoint *view;
float angle;
```

viewpoint_setviewangle is used to set the viewpoint's view angle.The view angle is defined as half the horizontal angular field of view (in radians). This is illustrated in Figure 5.1. The angle specified must be between 0.0 and PI/2.0 or the function will return with no effect.

When the view angle is set with viewpoint_setviewangle, this automatically sets the vertical view angle too, because setting the viewpoint's view angle determines the distance of the "eye" in front of the screen, which in turn determines the angle between the line to the center of the screen from the eye and a line to the top or bottom of the screen from the eye.

viewpoint_getviewangle

```
float viewpoint_getviewangle(view)
Viewpoint *view;
```

viewpoint_getviewangle returns the viewpoint's current view angle. This value is specified in radians.

viewpoint_sethithervalue

```
void viewpoint_sethithervalue(view, d)
Viewpoint *view;
float d;
```

viewpoint_sethithervalue is used to set the viewpoint hither clipping value. The default hither clipping value is 0.05 units, a small distance in front of the viewpoint. The new value d must be greater than the floating point "fuzz" value used by WorldToolKit, or the function will leave the hither clipping plane as close to the viewpoint as it can. See the section on "Roundoff and scaling" in Chapter 11 for more information on floating point tolerances in WorldToolKit.

viewpoint_gethithervalue

```
float viewpoint_gethithervalue(view)
Viewpoint *view;
```

viewpoint_gethithervalue returns the viewpoint's current hither clipping plane value (see Figure 6.1a).

viewpoint_setstereo

```
void viewpoint_setstereo(view, onoff)
Viewpoint *view;
FLAG onoff;
```

viewpoint_setstereo is used to turn the viewpoint's stereo parameter on or off. Stereo will be turned on by default if two DVI boards are found with properly installed VRAM drivers when universe_new is called. If only one DVI board is present in the system, this function call has no effect. See Chapter 12 for more information on stereoscopic display, and the Installation Guide for details of setting up your system to support twin DVI boards.

viewpoint_getstereo

```
FLAG viewpoint_getstereo(view)
Viewpoint *view;
```

viewpoint_getstereo returns TRUE or FALSE, depending on whether the viewpoint has been set to stereo.

viewpoint_setparallax

```
void viewpoint_setparallax(view, parallax)
Viewpoint *view;
float parallax;
```

viewpoint_setparallax sets the parallax value for stereo viewing. Parallax is the distance in the 3D virtual world in world coordinates between the points from which the left and right eye images are drawn.

viewpoint_getparallax

```
float viewpoint_getparallax(view)
Viewpoint *view;
```

viewpoint_getparallax returns the viewpoint's current parallax value.

viewpoint_setconvergence

```
void viewpoint_setconvergence(view, convergence)
Viewpoint *view;
short convergence;
```

viewpoint_setconvergence sets the stereo convergence value. Convergence is the horizontal offset in pixels between the left and right eye images. Convergence will have to be set to achieve stereo fusion in headmounted displays where the display screens are not exactly centered in front of the user's eyes. A negative convergence value moves the images for the eyes closer together, a positive value moves them further apart.

viewpoint_getconvergence

```
short viewpoint_getconvergence(view)
Viewpoint *view;
```

viewpoint_getconvergence returns the viewpoint's stereo convergence value, in screen pixel units.

7 Animation Sequences

Introduction to animation objects

WorldToolKit provides several mechanisms for displaying dynamically changing 3D objects:

1. Objects can be moved, using object_move, object_rotate, or other "move" functions, or by having a sensor attached to them.
2. Objects can be scaled or stretched (see object_scale and object_stretch).
3. Objects can be textured, and the applied textures can be modified.
4. Objects can be represented by an animation sequence, which is actually a collection of objects of varying form.

An animation sequence is composed of a set of different (although frequently similar) 3D objects which are sequentially added to and removed from to the universe to create a changing scene. Animation sequences can be used to represent a part as it is being assembled on an assembly line, a changing biological form, or any object or scene whose structure evolves in time.

The sequence of "keyframe" objects (to borrow a term from traditional animation) is constructed from individual model files. The graphical objects constructed from these files are not required to have any particular relationship to each other. However, proximity and similarity of consecutive objects determines the appearance of smoothness of the animation sequence.

Keyframe objects in an animation sequence are each stored as graphical objects (see Chapter 2). All of the functions pertaining to graphical objects can be applied to these keyframe objects, with the exception of object_add, object_remove, and object_delete. The application of these three functions to keyframe objects is managed by the animation sequence as a whole. The number of keyframe objects in an animation sequence and the complexity of these objects is limited only by the amount of RAM in your computer.

Animation sequence construction and destruction

animation_new

```
Animation *animation_new(filename, nframes, time, fast)
char *filename;
short nframes;
double time;
FLAG fast;
```

animation_new creates a new animation sequence and adds it to the universe. The animation sequence consists of a sequence of graphical objects which are loaded in from a sequence of files named filename.0, filename.1, ..., filename.(nframes-1). See the description under the function object_new for information on using the DOS environment variable WTMODELS to specify a path to the geometry files.

A time delay between frames, in seconds, is also specified. If the specified time turns out to be less than the minimum time to go through the simulation loop once, then this latter time will be used, in other words, the animation sequence would increment once each time through the simulation loop.

The last argument to animation_new is a FLAG specifying whether the keyframe objects created for this sequence should be rendered with the

faster rendering approach. If fast is TRUE, you may achieve a higher framerate at the expense of an occasional rendering error (that depends on the particular geometry in the virtual world). If fast is FALSE, rendering will be accurate but the framerate may not be as high. If the animation objects are the only graphical entities in the universe, then no rendering errors will be produced when fast is true. For the best framerate in this case, it is best to set fast to TRUE. See the discussion under object_new for more on the use of the fast FLAG.

For example:

```
Animation *a;
a = animation_new("myfile", 4, 0.25, TRUE);
```

creates an animation sequence from four files named myfile.0, myfile.1, myfile.2, and myfile.3. These four files must already exist at the time the call is made, and must be of one of the formats which can be read by the function object_new. A minimum time between frames of one quarter second is specified for this animation sequence. Finally, the faster rendering approach has been chosen for the objects in this animation sequence.

Any calls to animation_new must occur after the function universe_load is called (if universe_load is called by your program). This is because universe_load automatically calls universe_vacuum to make way for new graphics being loaded in. In addition, after all calls to animation_new are made, and before calling universe_go, you must call universe_ready to establish the colortable.

Since it takes time to load in a new animation sequence, animation_new should be called outside of the main simulation loop (i.e. before calling universe_go). If an animation sequence is not needed at the start of the simulation, you can call animation_remove to remove the animation sequence from the simulation, and then animation_add when the sequence is needed.

animation_delete

```
void animation_delete(animation)
Animation *animation;
```

animation_delete removes the animation sequence and its keyframe objects from the universe and frees them. The function universe_delete calls animation_delete, so at the end of your main program you do not need to call animation_delete if you call universe_delete.

animation_add

```
void animation_add(animation)
Animation *animation;
```

animation_add adds an animation sequence to the universe. When an animation sequence is created (with animation_new), it is automatically added to the universe. Therefore, animation_add is only needed to put an animation sequence which has been removed from the universe (with animation_remove) back into the universe.

When an animation sequence is added to the universe with animation_add, it is automatically started up, so that it is not necessary to call animation_go.

animation_remove

```
void animation_remove(animation)
Animation *animation;
```

animation_remove removes an animation sequence from the universe. The current keyframe object, that is, the animation sequence object which is currently in the universe, is also removed from the universe.

Animation sequence management

animation_next

```
Animation *animation_next(animation)
Animation *animation;
```

Use animation_next to iterate through the animation sequences currently in the universe. This does not include any animation sequences which have been removed from the universe with animation_remove.

animation_go

```
void animation_go(animation)
Animation *animation;
```

animation_go starts the animation sequence. The only time it is necessary to call animation_go is to restart an animation sequence that was previously stopped with a call to animation_stop. When an animation sequence is added to the universe (with animation_add), it is automatically started up.

animation_stop

```
void animation_stop(animation)
Animation *animation;
```

animation_stop stops the animation sequence at the current keyframe object. That object remains in the universe until animation_go is called.

An example of using animation_stop is given with the function animation_getnframes.

animation_getobject

```
Object *animation_getobject(animation, frame)
Animation *animation;
short frame;
```

animation_getobject returns the keyframe object which is number frame in the animation sequence.

The following example shows how to access the current keyframe object:

```
Animation *a;
Object *object;
object = animation_getobject(a, animation_getframe(a));
```

animation_setframe

```
void animation_setframe(animation, framenumber)
Animation *animation;
short framenumber;
```

animation_setframe sets the animation frame number to the value passed in. If framenumber is equal to or larger than the total number of key frames in the sequence, then the remainder of framenumber when divided by the total number of frames is used.

The following example increments the animation sequence frame number by 2.

```
Animation *a;
animation_setframe(a, animation_getframe(a) + 2);
```


animation_getframe

```
short animation_getframe(animation)
Animation *animation;
```

animation_getframe returns the current frame number for the animation. This frame number is one of 0, 1, ..., nframes-1, where nframes is the total number of keyframe objects as passed in to animation_new.

animation_getnframes

```
short animation_getnframes(animation)
Animation *animation;
```

animation_getnframes returns the total number of keyframe objects in the animation sequence. This number is the same as the number nframes passed in to animation_new.

In the following example, an animation sequence is stopped if it has reached the last keyframe object.

```
Animation *a;
if ( animation_getframe(a) == animation_getnframes(a) - 1 ) {
    animation_stop(a);
}
```

If this code fragment appeared in the universe's action function (see universe_setactions), then each time through the simulation loop the program would test whether the animation sequence had reached the last keyframe object, and stop the sequence if it had.

animation_settime

```
void animation_settime(animation, time)
Animation *animation;
double time;
```

animation_settime sets the time delay between frames. The time is given in seconds.

animation_gettime

```
double animation_gettime(animation)
Animation *animation;
```

animation_gettime returns the time delay in seconds between frames.

animation_move

```
void animation_move(animation, delta)
Animation *animation;
Posn6d *delta;
```

Use animation_move to rotate and/or translate all of the keyframe objects with one call. The Posn6d called delta whose pointer is passed in contains the *change* in position and orientation of the keyframe objects (in world coordinates), rather than the absolute positions and orientations to which they are to be moved. Keyframe objects are each rotated about their pivot points.

animation_move moves an animation sequence's keyframe objects even if the animation sequence has been stopped (with animation_stop) or removed from the universe (with animation_remove).

8 Portals

Introduction to portal objects

WorldToolKit contains a facility called "portals", which can be thought of as the 3D equivalent of links in a hypertext system. Portals are doorways in space which connect the currently displayed universe with an alternate universe. Only one universe is ever displayed at a time, and crossing a portal causes the connected universe associated with the portal to be displayed.

The judicious use of portals to structure your virtual environment into many connected universes can greatly speed display, since only objects in the current universe need ever be considered by WorldToolKit for rendering. Using portals also enables modular construction of virtual worlds, so that, for example, world-building can be divided up among several people.

For instance, the interior and exterior of a house model may be different universes, with the door to the house established as a portal which connects the two universes. When the viewpoint is outside the house, none of the features of the interior are considered for rendering. When the viewpoint crosses through the door, the universe for the house interior is current, with the outside detail of the house ignored.

Each portal is associated with a polygon at the time of portal creation. When the viewpoint is moved through the plane of the polygon associated with a portal and within the polygon extents, the connected universe for that portal is automatically loaded. Frequently, it may be desirable to associate portals to the same universe with all polygons in

an object. In this way, the new universe will be loaded whenever the viewpoint crosses into the interior of the object. A universe can have any number of portals to other universes (including itself). Portals are unidirectional. That is, they take you from the current universe to the connected universe, but not back. If traversal back to the original universe is desired, a second portal from the other universe to the original one can be created.

A portal associated with a polygon of an object moves with the object. If the object is copied, a copy of the portal is made. Stationary portals can be created as features of the background universe, as loaded with the universe_load command. If a moving object loaded by call to object_new contains polygons which are portals, those polygons will move properly with the object.

In the current version of WorldToolKit, portals are not saved with objects by the object_save call. Applications that desire portals to be persistent across runs must explicitly take care to save and restore portals.

Thought must be given to the placement of portals in an application. In the house example above, if only the door to the house is a portal and the house is entered by the viewpoint traversing a wall, no interior for the house will be seen. Portals are not very computationally intensive. In this example, one solution might be to make all walls of the house portals to the interior.

Portal construction and destruction

portal_new

```
Portal *portal_new(poly, universename)
Poly *poly;
char *universename;
```

portal_new creates a new portal object and adds it to the universe currently being displayed.

It takes two arguments: the polygon used to describe the position, orientation, and shape of the portal, and the filename of the universe to be loaded when the portal is crossed. Handles to polygons may be obtained from the universe_pickpolygon function.

For example, to select a polygon at screen coordinate (10.0,12.0) and use it to create a portal to a universe named "foo":

```
Posn2d pt;
Poly *p;

/* find polygon at screen point (10,12) */
pt[X] = 10.0; pt[Y] = 12.0;
p = universe_pickpolygon(pt);

/* only make portal if a polygon was found */
if ( p ) {
    portal_new(p, "foo");
}
```

Portals may also be created implicitly through their specification in a CAD file as described below.

Any calls to portal_new must occur after the function universe_load is called (if universe_load is called by your program). This is because universe_load automatically calls universe_vacuum to make way for new graphics being loaded in.

portal_delete

```
void portal_delete(portal)
Portal *portal;
```

portal_delete destroys a previously-created portal and removes it from the universe. If an object with polygons implicitly associated with portals is destroyed, as by call to object_delete, the associated portals are destroyed.

Creating portals implicitly

Due to the inconvenience of accessing individual polygons of an object once an application is running, it is often easier to specify portals as part of the file storing a model. Portals are implicitly created when models are loaded with universe_load or object_new if the geometry file has been appropriately annotated with portal information. The conventions for such annotation differ for the different file formats read by WorldToolKit.

For AutoCAD DXF files, the layer name is overloaded with portal information. Any layer name containing a "-" causes portals to be created for all polygons generated from all AutoCAD entities on that layer. The universe one reaches on crossing any portal for this layer is given by the name following the "-". For instance, a layer name "DOOR-GARDEN" causes all entities on that layer to be portals to a universe "GARDEN". When the viewpoint crosses any of the polygons built from these entities, the universe GARDEN is loaded. This

presumes the file GARDEN is accessible from the directory in which WorldToolKit is running and is in a form compressible to WorldToolKit, just as if it were are argument to the universe_load function. The DOS environment variable WTMODELS can specify a path to directories containing geometry to be loaded.

Note that if a "-" unintentionally occurs in a DXF layer name, WorldToolKit will interpret the string to the right of the "-" as the name of a universe to be accessible through a portal, and attempt to load a universe of that name when the portal is traversed. Be sure that layernames containing "-" do not appear in your DXF files, except when you wish to specify a portal.

For the WorldToolKit neutral ASCII file format as described in Appendix D, polygons to be used as portals are specified by the addition of a text string beginning with "-" to the line containing the list of vertices for the polygon. For instance, a polygon specification line of:

```
8 2 5 8 11 14 17 20 23 0xfff both    _V_PALEN2    -GALLERY
```

denotes a polygon which is to be a portal to a universe "GALLERY".

Portals may also be created explicitly through the WorldToolKit function portal_new. A pointer to the polygon required as an argument of portal_new can be obtained from the return value of the universe_pickpolygon function.

Crossing a portal

WorldToolKit checks each time through the simulation loop to see if any portals in the current universe have been crossed. If a portal has been crossed, then simulation manager automatically:

1. calls the exit function for the universe the viewpoint was in,
2. calls universe_load to create the new universe associated with the portal, (the call to universe_load implicitly invokes the entry function for the universe just entered, if any has been specified)

3. calls universe_ready to establish colors and universe geometrical parameters,
4. calls viewpoint_move, moving the viewpoint to the Posn6d derived from the call to universe_load. (For example, if the universe being entered originates from a DXF file, then the viewpoint is automatically moved to the viewpoint stored with this file. This data can be accessed with the calls viewpoint_getposition, or viewpoint_getorientation.),

The entry and exit functions referred to in the first and last items above are user-supplied functions which are invoked when a portal is crossed. As discussed in Chapter 1, the function universe_set_entryfn function can be used to specify a function to be called when a universe of given name is entered. Similarly, universe_set_exitfn is used to supply a function to be called when a universe is exited.

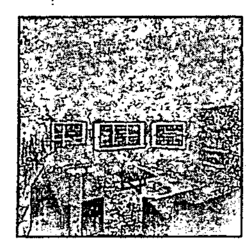
If for some reason two portals are crossed in the same iteration of the WorldToolKit simulation loop, then the simulation will proceed as if you had crossed the most recently created of the portals, entering the universe associated with it. If this situation does arise, it may mean that the spatial density of portals in the universe is too high for the speed at which the viewpoint is moving.

As a convenience, lights are preserved when portals are crossed. If you wish to have different lights in the new universe, your universe entry function could call lights_deleteall to delete the old lights, and then light_new or lights_read to create new lights.

9 Textures

Introduction to textures

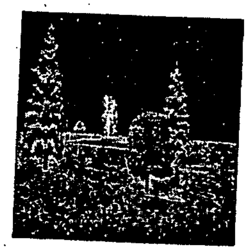
WorldToolKit supports a special capability which can greatly add to the realism of graphical objects. Surfaces of objects in the real world are not stark and cartoon-like, but have pattern, grain, and detail. To emulate this, polygons in WorldToolKit can be given a surface texture - a bitmap which is "stuck on" to the surface of the polygon and transformed with it. For example, think of depicting a table top with a uniform brown shaded polygon versus a table-top with an actual wood-grain image mapped onto it. A texture can be natural (video-derived) or synthetic (created in a paint program). The texture can look very authentic because it can be obtained from a video image of a real piece of wood. Floors can sport carpets and trees can have leaves. A human image can adorn a single textured polygon.



Judicious use of textures can result in far more complex and pleasing environments, without any of the overhead of modelling surface details or the runtime performance overhead of transforming details that are better "painted on" as textures. For example, instead of modelling as 3D details all of the windows of a distant building, a digital image of a real building can be applied to a single polygon which then serves as an entire side of the building. Modelling labor is conserved and rendering speed increases dramatically beyond what would have been necessary to model all of the details in 3D.

Textures are automatically transformed with the polygons to which they are applied, displaying perspective shift and scaling appropriate for the viewing parameters. WorldToolKit has functions for changing the orientation, scale, and offset of applied textures.

Textures may optionally be applied as shaded or transparent. Textures can not at present be both shaded and transparent. An ordinary texture retains the same pixel colors regardless of the relationship of the polygon to which it is applied to lights in the universe. Shaded textures appear at full brightness only when saturated with sufficient light, and diminish in brightness as do untextured shaded polygons. Transparent textures are applied only where source pixels are not entirely black.



Entirely black pixels are not transferred from the texture source. Regions of black pixels therefore appear as transparent. What was in view before transparent portions of textures are applied can still be seen through the transparent regions of the texture. The use of transparency adds realism to objects such as trees, where objects may be seen through the leaves. Ordinary textures are rendered significantly faster than either shaded or transparent textures. If performance is important, use the shading and transparency options only where necessary.

In some circumstances, such as entering a new universe through a portal, it may be desirable to minimize the amount of time required to load textures. For fast texture loading, textures may be loaded from RAM disk, with the DOS environment variable VIM set to refer to this disk. Files containing textures to be applied must always be located in either the current directory, or in one of the image directories specified in the paths given in the variable.

Texture format

To be used as textures in WorldToolKit, source bitmaps must be in DVI ".i16" format, at 128x120 resolution. The .i16 format is a true color 16 bit-per-pixel packed "YUV" format, with 6 bits of luminance information, and 5 bits each of the two color space dimensions. A complete description of the file format can be found in the Intel Actionmedia 750 Software Library Overview manual.

Included with WorldToolKit is an image conversion utility called VIMCVT.EXE. Documentation on this can be found in Appendix F. A DVI board must be installed in the system and the DVI drivers loaded to use this utility. This utility will convert from TARGA 16, 32 or .WIN files (these must not be compressed or RLE files) to the various DVI formats including .i16. A typical TARGA image would be converted to

a texture that WorldToolKit understands by the following command:

```
vimcvt -s128,120 targapic.t16 t16 16 dvipic.i16
```

where:

-s128,120	Scale the entire TARGA image to 128,120 pixels as required for WorldToolKit textures.
targapic.t16	The original TARGA 16 bit image.
t16	Input file format (TARGA 16, uncompressed).
16	Output file format (DVI .i16).
dvipic.i16	Output file name (texture name).

VIMCVT also supports the ability to scale and crop images. Textures can be created and modified with programs such as Adobe PhotoShop on the Macintosh, converted to TARGA format, and then used as textures with WorldToolKit. Many image processing tools on both the PC and the Macintosh support the TARGA format. Crystal (v3.52) on the PC can photorealistically render a 3D model, generate a TARGA image, and then this can be converted and used with WorldToolKit. Interesting effects can be created this way. For example, textured surfaces of models with shadows and highlights can be created with these programs and then loaded into WorldToolKit for use as textures in real time applications.

Another utility called DVIDUMP.EXE found in the util subdirectory can be used to capture images off the DVI screen. For example, suppose you want a texture of one of the scenes rendered by WorldToolKit so that you could use that image on a portal. You would do this by flying around until you had the exact image on the DVI screen that you want to convert to a texture. You would then quit out of WorldToolKit (without affecting the image), run the DVIDUMP utility and use the "save screen to file" feature to capture the image to a file. Finally, use VIMCVT to scale it for use as a texture for WorldToolKit.

Texture application

Textures may explicitly be applied to polygons or removed through the use of the following functions. The universe_pickpolygon function can be used to obtain a handle to the polygon to be textured. Sample code provided with WorldToolKit, in the module texture.c, illustrates the use of these texturing functions. Syntactic hooks into a variety of geometry file formats also permit the application of textures from within popular modelers, as described later in the section "Assigning textures implicitly".

See also the functions object_texture_apply and object_texture_delete in Chapter 2. These functions enable you to apply a texture to, or remove the textures from, all of an object's surfaces at once.

poly_texture_apply

```
FLAG poly_texture_apply(poly, filename, shaded, transparent)
Poly *poly;
char *filename;
FLAG shaded;
FLAG transparent;
```

This function applies a texture bitmap stored in a file with name filename to polygon poly, as well as to all polygons that are adjacent and coplanar with and have the same color as poly. (Therefore, if you have coplanar polygons to which you wish to assign different textures, these polygons must initially have different colors.) For the purpose of texturing, all coplanar adjacent polygons are considered part of the original polygon, and will in this discussion be referred to simply as "the polygon". If the polygon already had a texture applied, the old texture is replaced by the new texture. The FLAG arguments indicate whether the texture is to be shaded or transparent. Presently, textures cannot be both shaded and transparent. If both flags are given as true, transparency will be used.

The texture bitmap must be in DVI ".i16" format as described above, at 128x120 resolution. It is searched for in the current directory, and along the path given by the VIM environment variable in DOS.

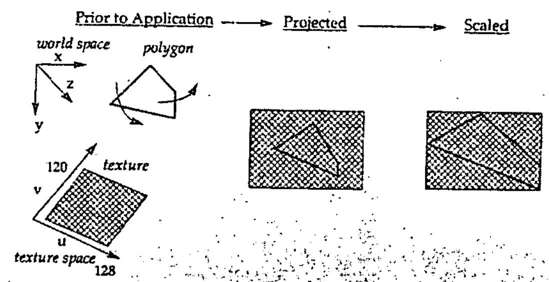


Figure 9.1 Texture Scaling

The initial rotation, scale, and offset of the texture are determined as follows (see Figure 9.1). The 2D source texture space has (u,v) texture coordinates. A polygon is specified in world 3D (x,y,z) coordinates. The polygon is rotated until it is parallel to the world (x,y) plane, the rotation being around the mean (center of gravity) of the polygon, vertex points. The polygon is then scaled independently in the x and y dimensions, so that its x coordinate lies between 0 and 127 (the maximum u for 128x120 textures) and its y coordinate lies between 0 and 119.

The function returns TRUE if the texture could be applied. In situations where the texture could not be found, as when the file named filename is not found, the function returns FALSE.

After application through this default mapping, textures may be modified through use of any of the texture modification functions: poly_texture_rotate, poly_texture_scale, poly_texture_translate, poly_texture_mirror described in the "Texture manipulation" section below.

See also the function object_texture_apply.

poly_texture_delete

```
void poly_texture_delete(poly)
Poly *poly;
```

poly_texture_delete removes a texture from a polygon that had previously been textured. If the polygon is not currently textured, this function has no effect.

See also the function object_texture_delete.

Assigning textures implicitly

Textures may be assigned to polygons implicitly, in a fashion similar to the implicit creation of portals by assigning them to polygons in the 3D model file prior to loading the file into WorldToolKit. The conventions for such annotation differ for the different file formats read by WorldToolKit. For AutoCAD DXF files, the layer name is overloaded with texture information. Any layer name beginning with the underscore character "_" is taken to be the name of the texture file to be applied to all polygons in that layer. The next character following a leading "_" in the texture name *must* be one of "V", "S", or "T" to signify a plain vanilla, shaded, or transparent texture. The third character in the layer name must be another "_", and the remainder of the string the name of the file containing the bitmap for the texture, in DVI ".i16" form. For instance, all polygons on a layer "_T_TREE23" will

have the transparent texture found in the file "TREE23" applied when the DXF file containing the layer is loaded into WorldToolKit.

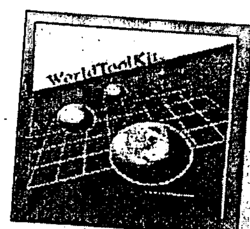
For the WorldToolKit neutral ASCII file format, polygons to be textured are specified by the addition of a text string with identical connotation to that used as an AutoCAD layer name. A syntactic convention similar to the AutoCAD DXF file above is used to specify the type of texture applied. For example, a polygon specification line of:

```
8 2 5 8 11 14 17 20 23 0xfff both _V_PALEN2 -GALLERY
```

denotes a polygon to be textured with a plain vanilla (unshaded) texture from the bitmap in a file PALEN2.

Texture manipulation

The initial application of a texture to a polygon is somewhat arbitrary, depending as it does on the orientation of the polygon in world coordinates. Once applied, textures may be modified in a number of ways by the following functions. An example of the use of these functions (through a simple keyboard-based interface) is supplied with WorldToolKit in source form as the wtk.c demo. The effect of the texture manipulation functions described below are described with respect to an initial texture:



It is not recommended that texture manipulation functions be used much within the real-time loop, since the texture copy with filtering which is performed to produce a tiled version of the texture when textures are scaled or rotated in certain ways is computationally intensive and can noticeably impair the frame rate.

poly_texture_rotate

```
void poly_texture_rotate(poly, angle)
Poly *poly;
float angle;
```

With this function, the texture on a polygon can be rotated in 2D (in texture space) on the surface of the polygon to which the texture is applied. angle specifies the amount of texture rotation in radians, around the "center of gravity" (arithmetic mean) of the vertices of polygon. Positive angles are counterclockwise rotations of the texture when the front face of the polygon is viewed.

Here is our sample texture rotated:

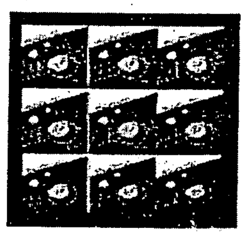
poly_texture_scale

```
void poly_texture_scale(poly, factor)
Poly *poly;
float factor;
```

Textures as applied to a polygon may be sized with this call. The argument factor gives the scale factor (applied homogeneously to the u and v texture coordinates). If factor>1.0, the texture coordinates are scaled up. That is, the texture bitmap size will be reduced on the surface of the polygon. If a scaling would cause any texture u,v coordinates to lie outside the extents of the polygon, the texture pattern is automatically tessellated by WorldToolKit. When factor<1.0, texture coordinates are scaled down, and the texture bitmap becomes larger on the surface of the polygon. Multiple levels of bitmap detail are retained as a texture is scaled up (and tesselated), so that if it is subsequently scaled down, the original bitmap is restored without loss of detail. As for rotation, scaling takes place about the center of gravity of polygon vertices.

Texture scaling resulting in tessellation of more than 32767 copies of a texture in either dimension may fail.

Here is our sample texture scaled down, with automatic tesselation:

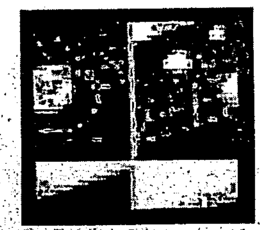


poly_texture_translate

```
void poly_texture_translate(poly, displacement)
Poly *poly;
Posn2d displacement;
```

Texture translation is used to shift the origin of the texture bitmap on the polygon surface - to "slide" the texture around. The displacement argument is a vector indicating how the applied texture is to be translated, in u,v space. Since textures are required to be 128x120 pixels, all translation in u is modulo 128, and that in v modulo 120. WorldToolKit permits a displaced texture to wrap at its edges in both dimensions. Note that applications of poly_texture_rotate cause a rotation of texture space, and that subsequent applications of this translation functions effect translation in the rotated space.

Here is our sample texture translated in both u and v:

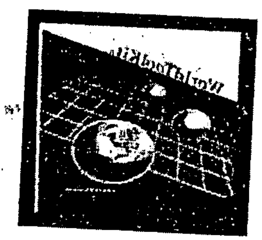


poly_texture_mirror

```
void poly_texture_mirror(poly)
Poly *poly;
```

One final function is required to permit complete control of textures. poly_texture_mirror "flips" an applied texture in 3D about the v axis of texture space. If you wish to mirror a texture about the u axis, use poly_texture_mirror to mirror it about the v axis, and then rotate the texture through PI/2 using poly_texture_rotate.

Here is the sample texture mirrored:



Texture paging

On a 2M DVI board, there are 32 pages of VRAM, of 64k each. The first 16 of these are used for the display buffers, (2 screens at 512x480x16 bits each). The final 3 pages are reserved for use by the DVI microcode. This leaves 13 pages of VRAM, each of which has space to store 2 bitmaps at 128x120x16 bits.

VRAM space for textures is a very limited resource, so a 2 stage caching and paging strategy is implemented in WorldToolKit to permit a very

large number of textures to be used in any universe by swapping texture bitmaps to a more slowly accessed area of VRAM and, if necessary, off the DVI card to protected-mode PC memory.

A total of 26 different textures can reside within VRAM on the DS1 DVI board. When more than 25 differently-textured polygons are present in the field of view at any time, paging off the DVI board to host memory occurs, significantly reducing the frame rate. It is recommended that the number of textures applied be limited so that this does not happen. If many different textures are desired, using them either in different universes, or on objects which tend to be seen individually is a prudent strategy from the standpoint of performance.

Note that when applied to a visible polygon a tessellated texture (as is automatically produced when a texture is scaled down) is stored in VRAM in its tessellated form for reasons of speed of application. Consequently, you must consider it as a distinct texture when computing the number of different textures that are visible.

10 Serial Ports

Serial port construction and destruction

serial_new

```
Serial *serial_new(port, intnum, configuration, bufsize)
unsigned short port;
short intnum;
unsigned char configuration;
short bufsize;
```

serial_new creates a serial port object. Once created, the serial port object can be passed in to the function sensor_new, when you wish to create a sensor object whose records are obtained by communication over the serial port.

The first argument to serial_new is the serial I/O port, and should be one of COM1, COM2 or COM3. Note that the I/O port mapping for COM3 is not standard on PCs, and it is advisable to use your own unsigned short constant for the value you know your COM3 I/O port to be mapped at, instead of using the COM3 defined constant supplied in WorldToolKit (0x3220).

The second argument is the hardware interrupt vector used for the serial port. Under PharLap 386|DOS-Extender Version 2.2, this defaults to 0x7c for COM1, and 0x7b for COM2. (Beware! Version 3.0 of the DOS-Extender uses the conventional DOS interrupts of 4 and 3 for COM1 and COM2 respectively.)

The third argument contains the configuration information. This is the bitwise OR of defined constants for the baud rate, parity, number of data bits, and number of stop bits. For example, configuration might be (BAUD96 | NP | DATA8 | STOP1) for a serial port configured for 9600 baud, no parity, 8 data bits, and 1 stop bit. The set of defined parameters is:

1. **baud rate.** BAUD192, BAUD96, BAUD48, BAUD24, BAUD12, BAUD60, BAUD30, BAUD15, and BAUD11 for: 19200, 9600, 4800, 2400, 1200, 600, 300, 150, and 110 baud rates, respectively. Note that the 19.2kb rate is non-standard and not supported by BIOS. It will probably work only for PCs with a 1.8432 Mhz crystal driving the UART divisor register.
2. **parity.** NP, OP, and EP for no parity, odd parity, and even parity, respectively.
3. **data bits.** DATA7 and DATA8 for 7 and 8 data bits, respectively.
4. **stop bits.** STOP1 and STOP2 for 1 and 2 stop bits, respectively.

An example of the use of serial_new is contained in WorldToolKit's geoball_new macro, which creates a Geometry Ball Jr. sensor object on COM1 or COM2.

```
#define geoball_new(port) \
    sensor_new(geoball_open, geoball_close, geoball_update, \
            serial_new(port, \
                        (port==COM1? 0x7c:0x7b), \
                        (BAUD96 | NP | DATA8 | STOP1), \
                        12))
```

Note that the interrupt values of 0x7c and 0x7b for COM1 and COM2 are for PharLap 2.2. Users of version 3.0 or greater of this DOS Extender should use the conventional DOS values of 4 and 3 respectively.

serial_delete

```
void serial_delete(serial)
Serial *serial;
```

serial_delete frees a serial port object. You will need to use this function only if you write your own sensor driver. serial_delete should be called only from the sensor's closefn.

Reading and writing to a serial port object

serial_read

```
short serial_read(serial, data, length, retry)
Serial *serial;
char *data;
short length;
FLAG retry;
```

serial_read reads a string of a specified length (number of bytes) from a serial port into the buffer called data. It returns the number of characters which were actually read. The length requested must be no larger than the (ring) buffer size for the serial port as the last argument to serial_new used in creating the serial port or a fatal error will result.

The retry flag set TRUE permits the serial_read call to wait until the requested number of characters have arrived. This can be useful if the serial_read calls are being made in such rapid succession that the requested number of characters have not had time to arrive. If after a large time out value (currently set to 3 seconds) the requested number of characters have still not arrived, a fatal error is generated.

serial_write

```
short serial_write(serial, buffer, length)
    Serial *serial;
    char *buffer;
    short length;
```

serial_write writes a string of a given length to a serial port and returns the number of characters which were successfully written. The argument buffer contains the string that is to be written to the serial port pointed to by serial port object serial, and length is the number of characters to be written. If the string is null-terminated, then length can be given as -1, and is computed from the string length of buffer.

11 Math Library

Introduction to the math library

WorldToolKit's math library contains functions for managing position and orientation data. The data types used in the math library are:

1.	Posn2d	2-dimensional vector (floating point)
2.	Posn3d	3-dimensional vector (floating point)
3.	Quat	quaternion (array of 4 floating point values)
4.	Posn6d	struct containing Posn3d and Quat
5.	Matrix3	3x3 array of floats
6.	Matrix4	4x4 array of floats

In WorldToolKit, orientation records are stored in quaternion form. If you prefer to work with matrices or euler angles, or if you are writing a sensor driver for a device which returns orientation records in one of these representations, then you will need to convert the records into the quaternion representation. Conversion functions are provided for going between matrix, euler angle, and quaternion representations of orientation.

It may be convenient, when indexing mathematical quantities in WorldToolKit, to use the constants X, Y, Z, and W, which have been defined as 0, 1, 2, and 3 respectively.

WorldToolKit math conventions

WorldToolKit's coordinate system obeys the right-hand rule. A viewpoint in the default orientation has the X axis pointing to the right, the Y axis pointing down, and the Z axis pointing straight ahead.

Rotations also obey the right-hand-rule. For example, if a vector pointing along the Z axis is rotated by 90 degrees about the X axis, it will end up pointing in the negative Y direction.

Rotation operators in WorldToolKit operate from right to left. What does this mean? Let's say you are working with an orientation that is represented by a 3x3 matrix. When this matrix multiplies a 3D vector, the 3D vector is rotated (hence the matrix is a "rotation operator"). There are two ways to multiply a vector and a matrix to create another vector. Either the vector is a column vector and the matrix is to its left (the matrix operates from left to right), or the vector is a row vector and the matrix is to its right (the matrix operates from right to left). Different results are obtained in these two cases. In WorldToolKit, matrices and quaternions follow the convention of operating from right to left. If you have a matrix which does not obey this convention (and it is a unitary matrix), call m3transpose or m4transpose to generate an acceptable matrix. Similarly, if you have a quaternion which does not obey this convention, call qinvert to generate an acceptable quaternion.

Posn2d: 2D vectors

A Posn2d is typedefed as an array of 2 floats.

p2copy

```
void p2copy(pin, pout)
Posn2d pin, pout;
```

p2copy copies pin into pout.

Posn3d: 3D vectors

A Posn3d is typedefed as an array of 3 floats.

pinit

```
void pinit(p)
Posn3d p;
```

pinit initializes a 3D vector so that p[X] = p[Y] = p[Z] = 0.0.

pcopy

```
void pcopy(pin, pout)
Posn3d pin, pout;
```

pcopy copies pin into pout.

pinvert

```
void pinvert(pin, pout)
Posn3d pin, pout;
```

pinvert negates a 3D vector, that is, sets

```
pout[X] = - pin[X];  pout[Y] = - pin[Y]; pout[Z] = -pin[Z];
```

pmag

```
float pmag(p)
Posn3d p;
```

pmag returns the length of a 3D vector.

add

```
void add(p1, p2, pout)
Posn3d p1, p2, pout;
```

add adds vectors p1 and p2 and puts the result in pout.

subtract

```
void subtract(p1, p2, pout)
Posn3d p1, p2, pout;
```

subtract subtracts vector p2 from p1 and puts the result in pout.

dot

```
float dot(p1, p2)
Posn3d p1, p2;
```

dot returns the dot product of p1 and p2.

mult_sv

```
void mult_sv(s, p)
float s;
Posn3d p;
```

mult_sv multiplies the vector p by the scalar (floating point value) s and leaves the result in p.

distance

```
float distance(p1, p2)
Posn3d p1, p2;
```

distance returns the distance between points p1 and p2.

light_getdirection

```
void light_getdirection(light, dir)
Light *light;
Posn3d dir;
```

light_getdirection stores the direction of a light in dir. This direction vector is normalized to have length equal to 1.0. (light_new normalizes the direction vector passed in.)

In the following example, a light is moved 10 distance units in the direction that it is pointing.

```
Light *light;
Posn3d dir, pos, newpos;

/* get the position and direction of the light */
light_getposition(light, pos);
light_getdirection(light, dir);

/* multiply the direction vector by 10.0 */
mult_sv(10.0, dir);

/* add the vector dir (which is now 10.0 units long) to
   pos to get the new light position. */
add(pos, dir, newpos);

/* move the light to newpos */
light_setposition(light, newpos);
```


light_setintensity

```
void light_setintensity(light, x)
Light *light;
float x;
```

light_setintensity sets the intensity of a light to x, which should be between 0.0 and 1.0. Since the computer can display colors only between a minimum and maximum intensity (which correspond to 0.0 and 1.0), any intensity values which are requested above or below this range are set to 1.0 and 0.0 respectively.

Examples of using light_setintensity are given under the functions light_next and light_getintensity.

light_getintensity

```
float light_getintensity(light)
Light *light;
```

light_getintensity returns a light's intensity value.

The following example uses the functions light_getintensity and light_setintensity to increase the intensity of a light by 5%.

```
Light *light;

light_setintensity(light, 1.05*light_getintensity(light));
```

5 Colors

Colortable optimization

The use of color on untextured object surfaces is managed in WorldToolKit with a 256 entry color lookup table. (The use of color on textured surfaces, which have 16-bit true color, is described in Chapter 9.) An algorithm for colortable optimization is used to select color entries for the table based on the colors present in models loaded by the application through calls to universe_load, object_new, animation_new, and the terrain functions. Given the frequency of occurrence of true colors present in the objects, color table entries are heuristically selected so as to minimize the misfit between the true colors requested and the colors actually displayed, while maximizing the size of the intensity ramp (number of intensity shades) for each color.

This optimization technique achieves pleasing results for most collections of graphical objects, especially if the number of different true colors present in the models being loaded is not unnecessarily large. Note that there is a trade-off between the number of different true colors requested, and the number of shades of each color available, since the colortable is a resource of limited size. For example, if a simple mechanical part containing polygons of only two colors is the only model loaded into the universe, then each color will have a (linear) ramp of 128 shades of intensity, from black to the requested color. The situation is not as simple in the case where a large number of true colors are requested. In this case, similar true colors are binned, and the color space recursively divided to produce a set of true colors representative

Quat: quaternions

A quaternion is a 4-component mathematical quantity which is the most efficient representation of orientations or rotations in 3D for many applications. The 4 degrees of freedom represented by a quaternion can be though of as representing a vector in 3D, and a twist around that vector.

In WorldToolKit, Quat is typedefed as an array of 4 floats. Quaternions in WorldToolKit are assumed to be normalized to have length equal to 1.0.

qinit

```
void qinit(q)
Quat q;
```

qinit initializes a quaternion, so that

```
q[X] = q[Y] = q[Z] = 0.0; q[W] = 1.0;
```

Suppose that q represents the orientation of a viewpoint in the world coordinate frame. The WorldToolKit convention is that when q is initialized as shown above, the viewpoint "looks" straight down the world Z axis, while the X axis points out to the right, and the Y axis points straight down.

qcopy

```
void qcopy(qin, qout)
Quat qin, qout;
```

qcopy copies qin into qout.

qinvert

```
void qinvert(qin, qout)
Quat qin, qout;
```

qinvert inverts a quaternion, setting

```
qout[X] = - qin[X]; qout[Y] = -qin[Y]; qout[Z] = -qin[Z];
qout[W] = qin[W];
```

The inverse of a quaternion corresponds to an inverse rotation.

qmult

```
void qmult(q1, q2, qout)
Quat q1, q2, qout;
```

qmult multiplies quaternion q2 into q1 (with multiplication from the left, as per the WorldToolKit convention) and puts the result in qout. Multiplying quaternions corresponds to composing rotations.

qmultinv

```
void qmultinv(q1, q2, qout)
Quat q1, q2, qout;
```

qmultinv multiplies quaternion q2 into the inverse of q1 and puts the result in qout.

rotate

```
void rotate(pin, q, pout)
Posn3d pin, pout;
Quat q;
```

rotate rotates a vector pin by the rotation represented by a quaternion q, and puts the result in pout.

Posn6d: position/orientation struct

WorldToolKit defines the following typedef for 6D position/orientation records:

```
typedef struct posn6d {
    Posn3d p;
    Quat q;
} Posn6d;
```

Note that to pass a Posn6d to a routine in C, one must explicitly take the address of the struct using the & operator to pass it by address. The following two functions illustrate the passing in of a Posn6d by address.

pqinit

```
void pqinit(pq)
Posn6d *pq;
```

pqinit initializes the Posn6d struct pointed to by pq.

```
void pqinit(pq)
    Posn6d *pq;
{
    pinit(pq->p);
    qinit(pq->q);
}
```

pqcopy

```
void pqcopy(pqin, pqout)
Posn6d *pqin, *pqout;
```

pqcopy copies pqin into pqout.

Matrix3, Matrix4: 3D and 4D matrices

WorldToolKit supports the use of 3D and 4D matrices. A 3x3 array of floats is typedefed as a Matrix3, and a 4x4 array of floats is typedefed as a Matrix4.

m3init, m4init

```
void m3init(m)
Matrix3 m;

void m4init(m)
Matrix4 m;
```

m3init and m4init (for 3D and 4D matrices respectively) set a matrix equal to the identity matrix.

m3copy, m4copy

```
void m3copy(min, mout)
Matrix3 min, mout;

void m4copy(min, mout)
Matrix4 min, mout;
```

m3copy and m4copy copy min into mout.

m3transpose, m4transpose

```
void m3transpose(min, mout)
Matrix3 min, mout;

void m4transpose(min, mout)
Matrix4 min, mout;
```

m3transpose and m4transpose (for 3D and 4D matrices respectively) put the transpose of min into mout.

mult_vm3

```
void mult_vm3(pin, m, pout)
Posn3d pin, pout;
Matrix3 m;
```

mult_vm3 multiplies a 3D vector by a 3x3 matrix from right to left, and puts the result in pout.

mult_mm

```
void mult_mm(m1, m2, mout, d1, d2, d3)
float *m1, *m2, *mout;
short d1, d2, d3;
```

mult_mm performs the matrix multiplication mout = m1 * m2, where m1 is a d1xd2 array of floats, m2 is a d2xd3 array of floats, and mout has dimensions d1xd3.

Conversion functions

Conversion functions are useful for going between various representations for orientation.

matrix_2_quat

```
void matrix_2_quat(m, q)
Matrix3 m;
Quat q;
```

matrix_2_quat converts a 3D matrix to a quaternion. The matrix is assumed to multiply vectors from the right.

12 System Issues

Stereoscopic output - convergence and parallax

The WorldToolKit library supports stereoscopic viewing using a pair of DVI boards, each generating a signal for one eye. See the WorldToolKit Installation Guide for details of how to configure your system to support two DVI cards. (The boards must be jumpered so their I/O port addresses do not conflict, and two VRAM.SYS drivers must be present in the CONFIG.SYS, with I/O ports and VRAM mapping pages set to non conflicting areas.)

If two cards are detected when the DVI graphics system is initialized (during call to universe_new), stereoscopic output will automatically be enabled. To display graphics only one DVI card even when two are installed in your system, the viewpoint_setstereo function can be called to set stereo FALSE for the universe's viewpoint:

```
viewpoint_setstereo(universe_getviewpoint(), FALSE);
```

At any time, one can query the universe's viewpoint to determine whether display is monoscopic or stereoscopic with the viewpoint_getstereo function.

Note that performance is better when the system is running in monoscopic mode. In order to generate a view for each eye with an appropriate parallax offset, each polygon in the viewing pyramid must be transformed twice when in stereo. This essentially halves system performance.

Adaptive screen resolution

The default screen resolution for WorldToolKit using the DS1 Actionmedia card is 256x240, 16 bits/pixel. It may be the case that during an iteration of the simulation loop, there is no change to the displayed view. This is the case when the viewpoint is stationary, no objects within the file of view are changing, and there is no change to the lighting. WorldToolKit notices when the displayed view is unchanging, and automatically doubles the spatial resolution in both dimensions, to 512x480. This provides a higher quality display, and the higher resolution is used until such a time as one of the above conditions ceases to be true.

Applications need not be aware of this adaptive change in resolution, since WorldToolKit changes its internal parameters as necessary to accommodate the change. The only difference you will notice is a filling in of "staircased" or aliased edges of objects as the resolution is doubled. Application-supplied convergence values as needed for tuning the interocular offset in head-mounted stereoscopic display outputs are automatically doubled or halved as the adaptive resolution changes occur, to maintain the separation of the right and left eye images. Applications should supply convergence values suited to the standard (256x240) screen resolution.

To force resolution to always be high or low or return to adaptive, the universe_setresolution function can be used.

13 Error Handling

Error handling in WorldToolKit

There are a number of conditions which may trigger fatal errors in WorldToolKit, such as running out of memory. A list of these is found in Appendix C. When such conditions are encountered, the function error is called, which prints a diagnostic message to the PC display screen and then calls universe_delete to clean up memory and terminate the graphics system prior to calling exit. The error function compiled into the WorldToolKit library is as follows:

```
void error(char *fmt, ...)
{
    va_list args;

    static inerror = FALSE;

    /* avoid recursive invocation */
    if ( inerror )
        return;

    fprintf(stderr, "WorldTool Error: ");
    va_start(args, fmt);
    (void) vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);

    inerror = TRUE;
```

```
        universe_delete();
        exit(-1);
    }
```

The error function is passed a varying number of arguments which consist of a format string supplied to vfprintf along with the subsequent arguments.

An application may wish to provide an alternative to the standard action of terminating the program when a fatal error condition is encountered. To do so, the application developer should provide their own version of error, and link it with their application to override use of the default error condition which would otherwise be found in the WorldToolKit library by the linker. It is the responsibility of this application-supplied error function to repair whatever condition led to the fatal error before returning to the code that called the error routine. (This may not always be possible.) In this version of WorldToolKit, application-provided error functions must analyze the string given as an argument to error in order to determine the cause of the error.

14 Performance Tips

Performance tips for WorldToolKit

Maintaining adequate frame-rate is essential for many applications constructed using WorldToolKit. The following rules of thumb can be useful in helping maximize system performance:

1. Structure your environment as different universes, connected by portals, limiting the number of polygons in each universe.
2. When your application environment consists of large spaces containing objects, some of which are viewed at a distance, use object_levelofdetail to produce objects at reduced levels of detail, and swap them in as the range to the object increases.
3. Stationary (background) objects loaded with universe_load will render much more quickly than moving objects. Try to limit the number of moving objects in your environment, and the polygonal complexity of the objects.
4. Use fast merging for moving objects which will not intersect other features of the environment, or when speed of moving objects is more important than perfect rendering. Fast merging is specified by setting the "fast" flag to TRUE in the call to object_new.
5. Shaded and transparent textures render more slowly than ordinary textures. Drawing a polygon with transparency is perhaps 20% slower than drawing the same polygon unshaded. Drawing a polygon with shading is perhaps 25% slower than drawing the same polygon unshaded.
6. Read textures from a RAM disk when it is important to reduce universe loading time.

Sensor constants

MOUSE_LEFTBUTTON
MOUSE_RIGHTBUTTON
MOUSE_MIDDLEBUTTON

SPACEBALL_BUTTON1
SPACEBALL_BUTTON2
SPACEBALL_BUTTON3
SPACEBALL_BUTTON4
SPACEBALL_BUTTON5
SPACEBALL_BUTTON6
SPACEBALL_BUTTON7
SPACEBALL_BUTTON8
SPACEBALL_PICKBUTTON

GEOBALL_LEFTBUTTON
GEOBALL_RIGHTBUTTON

Screen Resolution

RESOLUTION_LOW
RESOLUTION_HIGH
RESOLUTION_ADAPTIVE

Constraints

XCON
YCON
ZCON
XROTCON
YROTCON
ZROTCON

Default values

DEFAULT_VIEWANGLE
DEFAULT_HITHERVALUE
DEFAULT_AMBIENTLIGHT
DEFAULT_BGCOLOR

Serial port constants

COM1
COM2
COM3

BAUD192 (19.2 kb)
BAUD96 (9600 baud)
BAUD48 (4800 baud)
BAUD24 (2400 baud)
BAUD12 (1200 baud)

7. Use textured surfaces in place of modelled detail. Although textured polygons render more slowly than untextured polygons, the saving in total number of polygons in the model may be substantial when textures are used.
8. If your virtual world contains many different textures, follow the tips provided in Chapter 9 in the section called "Texture paging".
9. Use quaternions for the representation of orientation where possible. The multiplication of quaternions to compose orientations is faster than performing the equivalent operations using conventional rotation matrices.
10. Do not use too large a scale factor when loading models with universe_load or object_new. Using a scale factor which is too large (typically more than about 100.0) will cause WorldToolKit's renderer to slow down due to issues related to arithmetic precision. Performing any necessary scaling in your modelling program and using a scale factor of 1.0 when loading the geometry into WorldToolKit is optimal.

Several WorldToolKit functions may be helpful in tuning your application for maximum performance. These include universe_npolygons and object_npolygons, which return a count of the number of polygons present in the entire universe or in a specified object. Also of use may be universe_framerate, which returns the number of frames per second at which the application is running.

A

Defined Constants

Booleans

TRUE=1, FALSE=0

Mathematical constants

PI

INFINITY (=32767)

FUZZ (=0.004)

(X,Y, and Z are useful for indexing Posn3d's, while X,Y, Z, and W can be used to index Quat's)

X=0, Y=1, Z=2, W=3

U=0, V=1

Reference frames

FRAME_LOCAL
FRAME_WORLD
FRAME_VPOINT

BAUD60 (600 baud)
BAUD30 (300 baud)
BAUD15 (150 baud)
BAUD11 (110 baud)

NP (no parity)
OP (odd parity)
EP (even parity)

DATA7 (7 data bits)
DATA8 (8 data bits)

STOP1 (1 stop bit)
STOP2 (2 stop bits)

Screen constants

These screen parameters give the resolution of the image, and depend on whether the system is in high or low resolution mode. X screen coordinates are in the range [0, Width-1], where x==0 corresponds to the left edge of the screen. Y screen coordinates are in the range [0, Height-1], where y==0 corresponds to the top edge of the screen.

```
short Width, Height;
float Width2;     /* this is Width divided by 2.0 */
float Height2;    /* this is Height divided by 2.0 */
```


B Sample Application Code

A variety of sample WorldToolKit applications are provided on your distribution media in the "demo" subdirectory. This appendix contains other very simple examples.

Walkthrough program using a Geometry Ball

```
/*
 * This program loads in a model stored in the file "myfile", and lets you
 * navigate your viewpoint using a Geometry Ball device.
 */

#include <stdio.h>
#include "wt.h"

static Sensor *sensor;     /* the Geometry ball */
static Posn6d initial_p;   /* stores initial viewpoint */

/* The universe's action function is called each time through the
simulation loop. This particular action function looks for and acts upon
Geometry Ball button presses. A left button press ends the program
by calling universe_stop(). A right button press resets the viewpoint to
its original position and orientation by calling viewpoint_move(). */
static void actionfn();
```

```
main()
{
    /* Initialize the universe. (This must be the 1st WorldToolKit call.)*/
    universe_new();

    /* Load the graphical universe from the file "myfile". */
    universe_load("myfile", &initial_p, 1.0);

    /* Move the viewpoint to the position read from the model. */
    viewpoint_move(universe_getviewpoint(), &initial_p);

    /* Initialize a Geometry Ball sensor on serial port COM1. */
    sensor = geoball_new(COM1);

    /* Attach the sensor to the viewpoint. */
    viewpoint_addsensor(universe_getviewpoint(), sensor);

    /* Prepare the universe for start of the simulation. */
    universe_ready();

    /* Scale sensor sensitivity with the size of the universe. */
    sensor_setsensitivity(sensor, 0.01 * universe_getradius());

    /* Set the action function so that button presses will be acted on. */
    universe_setactions(actionfn);

    /* enter the main simulation */
    universe_go();

    /* all done; clean everything up. (This must be the last
    WorldToolKit call.) */
    universe_delete();

    return 0;
}
```



```
/*
 * Checks for button presses on the Geometry Ball.
 * Exits the simulation loop if the left button is pressed.
 * Restores the original viewpoint if the right button is pressed.
 */
static void actionfn()
{
    short buttons;

    /* get button press data from ball */
    buttons = sensor_getmiscdata(sensor);

    if ( buttons & GEOBALL_LEFTBUTTON ) {
        universe_stop();
    }
    else if ( buttons & GEOBALL_RIGHTBUTTON ) {
        viewpoint_move(universe_getviewpoint(), &initial_p);
    }
}
```

Program to preprocess a stationary backdrop

This example reads a geometry file whose name is provided as a command line argument, and saves the file (without a name suffix) as a preprocessed file in WorldToolKit internal form for subsequent fast loading.

```
#include <stdio.h>
#include <stdlib.h>
#include "wt.h"

static float scale = 1.0;

main(argc, argv)
    int argc;
    char *argv[];
```

```
{
    Posn6d x;
    FLAG ok;

    /* check for input and output file names */
    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s <input> <output>\n", argv[0]);
        exit(-1);
    }

    /* initialize the universe.
    (This must be the first WorldToolKit call in any application.) */
    universe_new();

    fprintf(stderr, "Converting...\n");

    /* process the file */
    if ( universe_load(argv[argc-2], &x, scale) ) {
        viewpoint_move(universe_getviewpoint(), &x);
        ok = universe_save(argv[argc-1]);
    }
    else {
        fprintf(stderr, "Error reading file %s\n", argv[argc-2]);
        ok = FALSE;
    }

    if ( ok )
        fprintf(stderr, "Conversion complete.\n");
    else
        fprintf(stderr, "Conversion failed.\n");

    universe_delete();

    return 0;
}
```


C Error Messages

In a number of circumstances, the program may terminate with an error diagnostic. Such fatal errors signify a condition which should not be allowed to arise in application code. The following list describes the reason for each of the more common error messages which can arise, and what corrective action can be taken. If you obtain an error message other than one listed here, it indicates an internal error. Please contact Sense8 for support in this case.

Already opened serial port

This message is produced by the function serial_new when a serial port is being opened, but has already been created by your application and has not (yet) been deleted by call to serial_delete. It is an error to try to open the same serial port twice.

Bird not responding

The Ascension Bird sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Bitmap chain corrupt?

This error is produced by call to poly_texture_delete when the last reference to a texture bitmap is being removed, and the texture can no

longer be found by the WorldToolKit texture manager. Seeing this error is probably symptomatic of memory allocation problems.

Couldn't find real mode server: realdvi.exe

Version 1.0 of WorldToolKit uses a graphics server running in real mode to communicate with DVI graphics routines. The executable for this server is named realdvi.exe, and must be on your DOS path or in the directory from which a WorldToolKit application is geing invoked when universe_new is called.

Couldn't open universe <universename>

In crossing a portal, the universe to be loaded could not be found. Ensure the universe <universename> exists, and is either in the current directory or in a directory on the path specified by the WTMODELS path. Note that the file must exist precisely as specified in <universename>, including any file type suffixes (such as .DXF) or lack thereof.

DVI error 0x<num1> errno <num2>

This is an internal DVI error message which should not arise. Please file a bug report, including the error numbers and as detailed a description of the conditions which produced the problem as you are able to provide.

Geoball not responding

The Geometry Ball Jr. sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable.

Incompatible realdvi.exe version

Version 1.0 of WorldToolKit uses a graphics server running in real mode to communicate with DVI graphics routines. The version of the real mode server must correspond to that of the WorldToolKit protected mode library. You will see this error if the two versions are not compatible.

Internal fault <num>

WorldToolKit has encountered a prohibited condition internally. Please file a bug report, including the error number <num> and as detailed a description of the conditions which produced the problem as you are able to provide.

Invalid reset message from Spaceball

The Spaceball has become confused. You should never see this message.

Invalid texture name: <texture name>

Texture names, when supplied implicitly as a DXF layer name or in the Sense8 neutral file format, must begin with one of the strings: "_S_", "_T_", "_V_" to signify that the texture is to be shaded, transparent, or plain vanilla. When supplied explicitly in a call to poly_texture_apply or object_texture_apply, the texture name should not contain the prefix, as texture type is given by other explicit arguments to the texture application function. Texture specification, either implicit within a model file or explicit in a call to a texture application function, which does not adhere to this syntactic requirement will result in the production of this error diagnostic.

Invalid line <num> in lights file <filename>

A syntactic error has been detected during reading of a file specifying lighting. The faulty line number is flagged in the message.

Layers_getcolor - unknown layer

During reading a DXF file, the DXF reader encountered a layer not specified in the layer table for the file. A syntactic error in the DXF file is suggested.

Load of cursor: Cursor.i16 failed

When you have specified a mouse sensor driver with a visible cursor, such as the mouse update function mouse_drawcursor, WorldToolKit looks for a 16x16 bitmap in .i16 form named cursor.i16 to use as a mouse cursor. A file by this name must be found either in the current directory, or in some directory on the path given by the DOS environment variable VIM.

Make_arc - too many verts in arc

There is currently a restriction of 512 vertices for any arc specified in a DXF file being loaded into WorldToolKit. Change your DXF file geometry to limit the number of vertices in any arc.

Mouse not found

WorldToolKit could not communicate with the mouse. Be sure your mouse driver is loaded, and that the mouse is properly cabled to your computer.

Must use version 2.13.12 DVI drivers

The DVI initialization invoked when universe_new was called found an inappropriate version of the DVI vram.sys driver(s) installed. The current release of WorldToolKit works only with the version 2.13.12 DVI drivers.

No DVI DS boards found

The DVI initialization invoked when universe_new was called failed to find any DVI hardware in your system. If you do have ActionMedia board(s) installed, this message may indicate a port or memory window conflict between your DVI hardware and other devices (such as a VGA card) in your computer. Check the arguments to the vram.sys DVI driver as described in the WorldToolKit installation manual. This message may also be seen if DVI cannot find the necessary system microcode. Be sure your DOS environment variable VVID points to the location of the DVI microcode, the default directory for which is \V\VID.

No serial port found

An attempt was made to open a serial port using the serial_new call, but the requested serial port hardware was not found. Be sure that the serial port you are trying to open exists on your computer.

Nonsensical scale factor

If a scale factor supplied to object_new is less than the limit of floating point precision supported by WorldToolKit, the program terminates with this error.

Not enough memory to run realdvi.exe

The real mode DVI graphics server, realdvi.exe, requires at least 400K of real mode memory to be able to run. Use your DOS "mem" command to check the amount of real mode memory available prior to invoking a WorldToolKit application. You may try removing device drivers and TSR programs, or use a memory-expander such as 386MAX or QEMM to obtain more real mode memory.

Object_new does not read files created with universe_save

You have tried to call the object_new function to load a dynamic object from a file which has been saved as a static background object by the universe_save call. You should be using the universe_load call to load such files.

OUT OF MEMORY!

An attempt to allocate more dynamic memory from the heap has failed, because your machine has exhausted its real mode and extended memory. Either you are loading models too big for the amount of physical memory in your computer, or are running other programs (such as a ram disk) which are occupying too much memory. Use your DOS "mem" command to check on the amount of extended memory available prior to invoking a WorldToolKit application.

Polhemus not responding

The Polhemus sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Polygon with fewer than 3 or more than 256 vertices specified

In Sense8's WorldToolKit neutral file format, polygons may not be degenerate; all polygons must have at least 3 vertices. 256 is the maximum number of vertices allowed.

Polygon vertices are not coplanar

In Sense8's WorldToolKit neutral file format, polygon vertices must lie in a plane.

Polyline has more than <num> vertices

There is a restriction on the number of vertices which can be present in any polyline entity in a DXF file being loaded into WorldToolKit. This is currently set to 512. Change your DXF file geometry to limit the number of vertices in any polyline.

Problem opening ucode file: <filename>

WorldToolKit uses special fast DVI microcode. The required microcode files must be found during the DVI initialization performed by call to universe_new. The necessary microcode binaries are provided in the UCODE subdirectory of the WorldToolKit distribution. Be sure either your DOS environment variable VVID includes this directory, or that you have copied the contents of this directory into the the default directory for DVI microcode, \V\VID.

Real mode error

This error will be seen following certain other WorldToolKit error messages. It indicates that the previous error occured in real rather than protected mode code, and may be ignored.

Real mode graphics server didn't open, code <num>

The real mode DVI graphics server, realdvi.exe, failed to open despite the fact that it was found. Either there is an invalid environment segment (indicating a memory corruption problem) or there is a problem with contents of the realdvi.exe file. Try restoring realdvi.exe from the WorldToolKit distribution media.

Scan.l - group code problem

The DXF parser has encountered a syntactic anomaly with the DXF file being read. Ensure the DXF file can be read by AutoCAD - the indication is that the contents of the file are corrupt.

Serial read - requested more than buffer size

When serial ports are created by call to serial_new, a buffer size is supplied to indicate the size of the input buffer to be created. An attempt to read more than this number of bytes by call to serial_read is not meaningful, and results in this error message.

Serial read timed out

Upon call to serial_read, a specified time can elapse for arrival of the requested number of bytes before the read fails with this error message, indicating the would-be transmitter is no longer alive. The timeout value is currently set somewhat arbitrarily to 3 seconds.

Spaceball not responding

The Spatial Systems Spaceball sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Syntax error

This error is produced by the DXF file reader when it encounters contents of the DXF file it cannot understand. Ensure the DXF file can be read by AutoCAD - the indication is that the contents of the file are corrupt or that it was generated by some third party DXF output program which does not generate legitimate DXF.

Universe_load does not read files created with object_save

You have tried to call the universe_load function to load a static background object from a file which has been saved as a dynamic object by the object_save call. You should be using the object_save call to load such files.

Unknown file type: <filename>

This message is produced by universe_load, object_new, or animation_new if an attempt is made to load a file which is not in one of the file formats understood by WorldToolKit.

Unknown serial port

WorldToolKit only supports 3 serial ports, COM1 through COM3. An attempt to call serial_new for any other serial port results in this error.

Unrecognized file type

WorldToolKit is able to read geometry files specified in a number of formats. This message indicates that the file passed to universe_load or object_new is not in one of the formats understood by WorldToolKit.

Vram.sys not running

The DVI initialization invoked when universe_new was called was unable to communicate with the vram.sys device driver(s) needed for DVI. Be sure that you have the vram.sys device drivers in your config.sys file, as documented in the WorldToolKit installation manual.

Write to serial port timed out

A problem was encountered during a serial_write. A hardware fault or memory corruption is indicated.

D

Writing a Sensor Driver

Writing a sensor driver in WorldToolKit consists of providing the arguments to the function sensor_new. sensor_new, which creates a sensor object, has the format:

```
Sensor *sensor_new(openfn, closefn, updatefn, serial)
void (*openfn)( ), (*closefn)( ), (*updatefn)( );
Serial *serial;
```

where

1. openfn is a function which initializes the device,
2. closefn is a function which closes the device and cleans up,
3. updatefn is a function which gets records from the device, and
4. serial is a serial port object as returned by the function serial_new.

If you do not wish to use WorldToolKit's prepackaged driver functions (the openfn, closefn, and updatefn functions provided for each sensor supported in WorldToolKit), or if you have a device which is not yet supported in WorldToolKit, then you will need to provide your own driver functions. This chapter is a specification for these functions.

Overview

WorldToolKit math conventions

The data read from your device must be made consistent with WorldToolKit's math conventions (see Chapter 11: Math library). In particular, you will need to keep in mind the following:

1. The position and orientation records from your device may have to be transformed to be consistent with WorldToolKit's coordinate convention.
2. In WorldToolKit, orientation records, including those stored with sensor objects, are stored in quaternion form. If you prefer to work with matrices or euler angles, or if your device returns orientation records in one of these representations, then you will need to convert these records into quaternion form as part of generating a new sensor record. Conversion functions matrix_2_quat and euler_2_quat are provided as part of the WorldToolKit math library.
3. Orientation records must be stored in such a way that they operate from right to left. If you have a matrix which does not obey this convention (and it is a unitary matrix), call matrix_transpose to generate an acceptable matrix. Similarly, if you have a quaternion which does not obey this convention, call qinvert to generate an acceptable quaternion.

Sensor records must be relative

In its present version, all devices in WorldToolKit are expected to generate relative position and orientation records. If your device returns absolute records, then the driver function updatefn will have to compute the change in position and orientation since the last time the sensor was read. WorldToolKit provides functions which simplify this task (see below).

Constraining sensor records

If you will want the ability to apply constraints to your sensor input (see the WorldToolKit function sensor_setconstraints), then your sensor driver, namely the function updatefn, should generate a sensor record that is consistent with the constraint flags set for the sensor.

There are 6 constraint flags, 3 for constraining translations (XCON, YCON, and ZCON) and 3 for constraining rotations (XROTCON, YROTCON, and ZROTCON). The example at the end of this chapter illustrates how to incorporate these constraints into a sensor driver.

Scaling sensor records

Two scale factors, one for translations and one for rotations, are stored in the Sensor struct. The WorldToolKit calls sensor_setsensitivity and sensor_setangularrate are provided so that these scale factors can be modified. It is convenient, for example, to be able to scale translational values with the size of the universe.

If you wish to take advantage of this feature when writing your sensor driver, then multiply the translational values returned by your device by the value returned by sensor_getsensitivity, and the angular values returned by your device by the value returned by sensor_getangularrate.

If your device returns absolute (rather than relative) records, then it may not be desirable to scale rotation records, although scaling translation records may still be useful. For example, suppose your device is an absolute sensor worn on the head, used to track viewpoint. In a realistic simulation, a 360 degree turn of the head should correspond to a 360 degree turn in the virtual world. If this is what is desired, then rotational input from the device should not be scaled by the value returned by sensor_getangularrate. Sensor input *will* still have to be relativized, however, as described in the section below on updatefn.

It may also be useful to scale input from the sensor by the largest value returned by the device. For translation records, then, the resulting scaled values would be in the range [-sensitivity, sensitivity], where sensitivity is the value returned by the function sensor_setsensitivity, in the same units as distances in the graphical world. The advantage of this is that sensitivity can then be interpreted as a maximum speed along any axis, as described under the function sensor_setsensitivity. The same applies for rotation records and the value returned by sensor_getangularrate.

When a new sensor object is created (with sensor_new), the following default values are set:

```
sensor_setsensitivity(sensor, 1.0);
sensor_setangularrate(sensor, PI/36.0); /* 5 degrees, in radians */
```

If sensor values are scaled in the manner described above, and the device is attached to the viewpoint object, then each time through the simulation loop, the viewpoint will translate at most 1 distance unit along any axis and will rotate at most 5 degrees about any axis. These rates can be changed with calls to sensor_setsensitivity and sensor_setangularrate.

The example at the end of this chapter illustrates how to incorporate these scale factors into a sensor driver.

Talking to the serial port

Many sensors are serial peripheral devices. WorldToolKit contains routines for reading and writing to serial ports. See Chapter 10 of the Reference Manual for more on this subject.

Include files

Your sensor driver should have the following include statement:

```
#include "wt.h"
```

Assuming your compiler can find the path to this include file supplied with WorldToolKit, all required typedefs (such as that for Sensor) should be found.

The driver functions

You will only need to refer to the driver functions openfn, closefn, and updatefn when you pass them in to the sensor object constructor function sensor_new. Other than that, you should not refer directly to the driver functions in your program.

openfn

```
void openfn(sensor)
Sensor *sensor;
```

The purpose of openfn is to initialize the device.

If the device you are using returns absolute position and orientation records, obtain the first sensor record and store it with the sensor object using the call

```
sensor_setlastrecord(sensor, p, q);
```

where p is of type Posn3d, and q is of type Quat (quaternion). Of course, p and q must be consistent with WorldToolKit's math conventions as described above. If your device returns orientation

records in either matrix form or as euler angles, then the functions matrix_2_quat or euler_2_quat can be used to obtain the corresponding quaternion q. (The absolute position/orientation record stored with the sensor object with the call sensor_setlastrecord is used in updatefn to generate a relative position/orientation record.)

Finally, if the device is to be polled each time through the simulation loop rather than streaming data continuously, then you should request the next record before exiting openfn.

closefn

```
void closefn(sensor)
Sensor *sensor;
```

If your device is a serial port device, then in this function, you may wish to retrieve any remaining data which has been sent from the device to the serial port. Then, free the serial port object with the call

```
serial_delete(sensor_getserial(sensor));
```

The function closefn is called by WorldToolKit when you call sensor_delete or universe_detete (which calls sensor_delete).

updatefn

```
void updatefn(sensor)
Sensor *sensor;
```

The purpose of updatefn is to obtain a new sensor record. This function is called automatically each time through the simulation loop by the WorldToolKit simulation manager for all sensors that have been added to the universe.

The function updatefn has the following four parts:

1. A new data record is obtained from the device.
2. The new record is used to generate relative position (p) and orientation (q) values, where p is a Posn3d and q is a Quat (quaternion). These values may be constrained, scaled, or both, as described in the sections "Constraining sensor records" and "Scaling sensor records".
3. p and q are stored with the sensor object by calling

   ```
   sensor_setrecord(sensor, p, q);
   ```

   p and q make up the new sensor record. If the sensor is attached to a graphical object, for example, then p and q are used to change the object's position and orientation.
4. If the sensor is a serial port device and is being polled, the next record is requested.

Steps 1 and 4 above require that you know how to "talk to" the sensor device. WorldToolKit provides utility functions for reading and writing to a serial port. (See Chapter 10 for a description of these functions.) Or, you may provide your own routines for communicating with the device.

Step 3 above simply involves calling sensor_setrecord exactly as shown above.

That leaves Step 2, that is, how to generate p and q from the data read from your device.

How p and q are generated from the data read from the device is really up to you. For example, your input device might generate only X and Y coordinate information and button presses. (This is what the typical mouse device returns.) Example 1 at the end of this Appendix is an example of a typical update function for such a device. This update function generates yaws from button presses, forward/back motion from Y screen values, and left/right motion from X screen values. (For more information specific to use of the mouse, see Chapter 3.)

If your device returns position or orientation records which are 3-dimensional, you may need to convert the data so that it is consistent with the WorldToolKit math conventions described above.

Then, if the resulting record is relative (that is, it corresponds to a change in position and orientation rather than an absolute position and orientation), you need only store this information in p and q and you are done with Step 2. If your orientation record is in matrix or euler angle form, then q may be obtained by calling matrix_2_quat or euler_2_quat.

If, on the other hand, the sensor record is absolute, you will need to turn it into a relative record. To do so, first convert the orientation record into a quaternion if not already stored that way. Then call:

```
sensor_relativizerecord(sensor, absolute_p, absolute_q, p, q);
```

where absolute_p and absolute_q are the absolute records passed in, and p and q are the relative records returned, which can then be passed in to sensor_setrecord.

Since the function sensor_relativizerecord uses the absolute sensor record from the last time through the simulation loop (which was stored with the call to sensor_setlastrecord), you will need to call these functions with the new absolute record, for use next time through the loop. In other words, after the call to sensor_relativizerecord, you should call

```
sensor_setlastrecord(sensor, absolute_p, absolute_q);
```

if your device returns absolute records.

Finally, you may wish to store other kinds of data such as button

presses with the sensor object. Such information might be used, for example, in the universe's action function as a trigger of activity. (See the function universe_setactions.) To store this data with the sensor, use the call

```
sensor_setmiscdata(sensor, x);
```

where x is a short. This data can then be retrieved with the call

```
x = sensor_getmiscdata(sensor);
```

If your driver is for a device supported in WorldToolKit, then you may wish to use the defined constants for button press and other data given in Appendix A.

Example 1: Update function for the mouse

```
/*
 * Example of a mouse update function.
 * This update function first obtains the raw screen coordinates
 * and button presses from the mouse device by calling
 * mouse_rawdata.
 *
 * The sensor translation record p is then computed, with X screen
 * values used to generate left/right motion, and Y screen values
 * used to generate forward/back motion.
 *
 *
 * The sensor rotation record is obtained from left and right button
 * presses.  Left button presses generate yaw left; right button presses
 * generate yaw right.
 */
void mouse_myupdate(sensor)
    Sensor *sensor;
{
```

```
#define ESC          0x1b  /* hex value of ESC character */
#define NBYTES       12    /* 12 bytes for pos/orientation record */
#define MAXGEOVAL 128      /* max value returned by geometry ball */

/* command buffer - sent to ball */
static char cmd[3] = {ESC};

/*
 * Initialize the geometry ball.
 */
void geoball_open(sensor)
    Sensor *sensor;
{
    /* set polling mode to request and request 2 bytes*/
    cmd[1] = 'R';
    serial_write(sensor_getserial(sensor), cmd, 2);

    /* set output mode to binary */
    cmd[1] = 'P';
    cmd[2] = 'C';
    serial_write(sensor_getserial(sensor), cmd, 3);

    /* set request byte, and request 1st record */
    cmd[1] = 0x05;
    serial_write(sensor_getserial(sensor), cmd, 2);
}

/*
 * All done; delete the geometry ball's serial port object
 */
void geoball_close(sensor)
Sensor *sensor;
{
    serial_delete(sensor_getserial(sensor));
}
```



```
/*
 * Acquire a new record from the geometry ball.
 * The geometry ball returns X, Y, and Z translation values,
 * X, Y, and Z rotation values, and left and right button presses.
 */
void geoball_update(sensor)
Sensor *sensor;
{
    Posn3d p;                        /* translational input */
    float wx,wy,wz;                  /* rotational input */
    Quat q;                          /* stored with sensor object */
    char geobal[NBYTES];             /* stores record from geometry ball */
    float trans_factor,ang_factor;   /* scale factors */
    short constraints;

    /* get record from serial port */
    serial_read(sensor_getserial(sensor), &geobal[0], NBYTES,TRUE);

    /* scale factors for geoball inputs */
    trans_factor = sensor_getsensitivity(sensor) / MAXGEOVAL;
    ang_factor = sensor_getangularrate(sensor) / MAXGEOVAL;

    /* what constraints are set? */
    constraints = sensor_getconstraints(sensor);

    /* scale and constrain translation record */
    if ( constraints&XCON )
        p[X] = 0.0;
    else
        p[X] = (float) geobal[4] * trans_factor;
    if ( constraints&YCON )
        p[Y] = 0.0;
    else
        p[Y] = (float) -geobal[5] * trans_factor;
    if ( constraints&ZCON )
        p[Z] = 0.0;
    else
```

```
float wy;                  /*to store yaw value, in radians */
short buttons;             /* stores button press data */
FLAG lbutton, rbutton;     /* left and right button presses */
Mouse_rawdata *pos;        /* raw mouse x,y record */
Posn3d p;                  /* for call to sensor_setrecord */
Quat q;                    /* for call to sensor_setrecord */
float speed;               /* sensor sensitivity value */
short constraints;         /* sensor constraints. */

/* get new raw data record from device */
mouse_rawdata(sensor);

/* get raw x and y mouse values in screen coordinates */
pos = (Mouse_rawdata *)sensor_getrawdata(sensor);

/* get sensor constraints */
constraints = sensor_getconstraints(sensor);

/* transform raw screen values to translation record.
Scale the values to lie between -speed and +speed. */
speed = sensor_getsensitivity(sensor);
pinit(p);
if ( ! (constraints & XCON) ) {
    p[X] = (pos->x - Width2) * speed / Width2;
}
if ( ! (constraints & ZCON) ) {
    p[Z] = (Height2 - pos->y) * speed / Height2;
}

/* which buttons were pressed? */
buttons = sensor_getmiscdata(sensor);
lbutton = buttons & MOUSE_LEFTBUTTON;
rbutton = buttons & MOUSE_RIGHTBUTTON;

/* generate yaw value, scaled by the sensor's angular rate */
if ( constraints & YROTCON ) {
    qinit(q);
```



```
    }
    else if ( lbutton && !rbutton ) {        /* left button press */
        wy = -sensor_getangularrate(sensor);
        euler_2_quat(0.0, wy, 0.0, q);
    }
    else if ( rbutton && !lbutton ) {        /* right button press */
        wy = sensor_getangularrate(sensor);
        euler_2_quat(0.0, wy, 0.0, q);
    }
    else {
        qinit(q);
    }

    /* store the record with the sensor */
    sensor_setrecord(sensor, p, q);
}
```

Example 2: Driver for the Geometry Ball Jr.

The Geometry Ball Jr. is a desktop device produced by CIS Graphics, Inc. It senses forces and torques and returns relative x, y, z, roll, pitch, yaw records.

```
/*
 * Interface to the Geometry Ball Jr.
 * initialization (geoball_open);
 * termination (geoball_close);
 * updating (geoball_update)
 *
 * Copyright (c) 1991 Sense8 Corporation
 */

#include <stdio.h>
#include <stdlib.h>
#include "wt.h"
```

```
        p[Z] = (float) -geobal[6] * trans_factor;

    /* scale and constrain orientation record */
    if ( constraints&PITCHCON )
        wx = 0.0;
    else
        wx = (float) geobal[7] * ang_factor;
    if ( constraints&YAWCON )
        wy = 0.0;
    else
        wy = (float) -geobal[8] * ang_factor;
    if ( constraints&ROLLCON )
        wz = 0.0;
    else
        wz = (float) -geobal[9] * ang_factor;

    /* convert eulers angles to quaternion */
    euler_2_quat(wx, wy, wz, q);

    /* store translation / rotation record with sensor */
    sensor_setrecord(sensor, p, q);

    /* store button presses */
    sensor_setmiscdata(sensor, geobal[3]);

    /* request next record */
    serial_write(sensor_getserial(sensor), cmd, 2);
}
```


E

Neutral ASCII File Format

Overview

The Sense8 neutral ASCII file format is a generic representation for polygonal geometry. The Sense8 WorldTool system can at present read geometry in any of the following file formats:

1. AutoCAD .DXF (through Release 11)
2. Sculpt-3d
3. Caligari
4. Videoscape

In order to import other geometry into WorldTool, an additional file format is supported, an ASCII representation of vertices and their connectivity to form polygonal faces. This Sense8 neutral ASCII file format serves as an interface between modellers which cannot write geometry in any of the other forms accepted by WorldTool. It is intended that users write translators to transform their proprietary format into the neutral ASCII file format, which can then be read directly into WorldTool.

In the ASCII file format, objects are represented as sets of polygons, and polygons are ordered collections of vertices. Polygons have colors and can optionally have other attributes specified. Objects can optionally be named.

The file must begin with a line containing the string "s8". (This is used by WorldTool to determine the type of the file.) There follows a set of one or more object specifications. All lines must be terminated by a carriage return. (The PC end-of-line convention is '\r \n', or in hex, 0xd and 0xa).

Each object specification starts with a line of text giving the object's symbolic name. The next line contains the number of vertices in the object. Vertex x,y,z coordinates, as real numbers, follow one per line. The next line contains the number of polygons in the object. Polygon specification lines follow, one for each polygon.

The Vertex coordinate lines should contain 3 real numbers (as could be read in C with a "%f %f %f" format string). One or more spaces must separate the numbers.

Each polygon specification line starts with an integer giving the number of vertices in the polygon. That number of indices follow, each indexing a vertex coordinate. (Zero designates the first vertex). After the vertex indices is a color designator, given in hexidecimal as a number in the range 0x0 to 0xfff. The high order 4 bits indicate the the red intensity, the middle 4 bits the green, and the low order 4 bits the blue intensity of the color for the polygon. The optional string "both" indicates that both sides of the polygon are to be visible. The default is only the front face, by the right hand rule. Optionally, a texture name can follow as the last field on a polygon specification line. (When texturing is on, color is ignored for textured polys; surface properties come from the texture).

Texture names give the file containing the bitmap to be used as a texture. Textures may optionally be shaded or transparent. Shaded textures have their brightness affected by the lights present in the model. Transparent textures are rendered so that all black pixels in the source bitmap are transparent when the texture is applied to a polygon.

Texture names begin with the character "_". The character following the "_" indicates the type of texture, according to the following:

_v_	plain Vanilla texture (no shading)
_s_	Shaded texture
_t_	transparent texture

For example, a texture named "_v_rug" causes a texture from a file named "rug" to be used. A texture named "_s_rug" would apply the same texture, but shaded based on lighting.

At present, these three texture options are mutually exclusive.

Sample ASCII File

The following is an example of a simple ASCII File containing a few simple geometric structures and archway. All polygons of the second cube are textured with a shaded texture from a bitmap file called "fabric". There follows a set of one or more object specifications. Each line is terminated by a carriage return. (On the PC this is generated with '\r \n').

```
s8
Cube
8
3.144832 -3.144832 2.656802
2.855168 -3.144832 2.656802
2.855168 -2.855168 2.656802
3.144832 -2.855168 2.656802
3.144832 -3.144832 0.513213
2.855168 -3.144832 0.513213
2.855168 -2.855168 0.513213
3.144832 -2.855168 0.513213
6
4 0 1 2 3 0xff0
```

```
4 7 6 5 4 0xff0
4 0 4 5 1 0xff0
4 1 5 6 2 0xff0
4 2 6 7 3 0xff0
4 3 7 4 0 0xff0
Cube,1
8
-2.855168 -3.144832 2.653589
-3.144832 -3.144832 2.653589
-3.144832 -2.855168 2.653589
-2.855168 -2.855168 2.653589
-2.855168 -3.144832 0.509999
-3.144832 -3.144832 0.509999
-3.144832 -2.855168 0.509999
-2.855168 -2.855168 0.509999
6
4 0 1 2 3 0xff0 _S_FABRIC
4 7 6 5 4 0xff0 _S_FABRIC
4 0 4 5 1 0xff0 _S_FABRIC
4 1 5 6 2 0xff0 _S_FABRIC
4 2 6 7 3 0xff0 _S_FABRIC
4 3 7 4 0 0xff0 _S_FABRIC
Cut-cone
32
3.000000 -3.250000 0.500000
2.905000 -3.227500 0.500000
2.825000 -3.175000 0.500000
2.772500 -3.095000 0.500000
2.750000 -3.000000 0.500000
2.772500 -2.905000 0.500000
2.825000 -2.825000 0.500000
2.905000 -2.772500 0.500000
3.000000 -2.750000 0.500000
3.095000 -2.772500 0.500000
3.175000 -2.825000 0.500000
3.227500 -2.905000 0.500000
3.250000 -3.000000 0.500000
```



```
3.227500 -3.095000 0.500000
3.175000 -3.175000 0.500000
3.095000 -3.227500 0.500000
3.000000 -3.500000 0.000000
2.810000 -3.455000 0.000000
2.650000 -3.350000 0.000000
2.545000 -3.190000 0.000000
2.500000 -3.000000 0.000000
2.545000 -2.810000 0.000000
2.650000 -2.650000 0.000000
2.810000 -2.545000 0.000000
3.000000 -2.500000 0.000000
3.190000 -2.545000 0.000000
3.350000 -2.650000 0.000000
3.455000 -2.810000 0.000000
3.500000 -3.000000 0.000000
3.455000 -3.190000 0.000000
3.350000 -3.350000 0.000000
3.190000 -3.455000 0.000000
18
16 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  0xff0
16 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16  0xff0
4 0 16 17 1  0xff0
4 1 17 18 2  0xff0
4 2 18 19 3  0xff0
4 3 19 20 4  0xff0
4 4 20 21 5  0xff0
4 5 21 22 6  0xff0
4 6 22 23 7  0xff0
4 7 23 24 8  0xff0
4 8 24 25 9  0xff0
4 9 25 26 10  0xff0
4 10 26 27 11  0xff0
4 11 27 28 12  0xff0
4 12 28 29 13  0xff0
4 13 29 30 14  0xff0
4 14 30 31 15  0xff0
```

```
4 15 31 16 0 0xff0
Cut-cone,1
32
-3.000000 -3.250000 0.500000
-3.095000 -3.227500 0.500000
-3.175000 -3.175000 0.500000
-3.227500 -3.095000 0.500000
-3.250000 -3.000000 0.500000
-3.227500 -2.905000 0.500000
-3.175000 -2.825000 0.500000
-3.095000 -2.772500 0.500000
-3.000000 -2.750000 0.500000
-2.905000 -2.772500 0.500000
-2.825000 -2.825000 0.500000
-2.772500 -2.905000 0.500000
-2.750000 -3.000000 0.500000
-2.772500 -3.095000 0.500000
-2.825000 -3.175000 0.500000
-2.905000 -3.227500 0.500000
-3.000000 -3.500000 0.000000
-3.190000 -3.455000 0.000000
-3.350000 -3.350000 0.000000
-3.455000 -3.190000 0.000000
-3.500000 -3.000000 0.000000
-3.455000 -2.810000 0.000000
-3.350000 -2.650000 0.000000
-3.190000 -2.545000 0.000000
-3.000000 -2.500000 0.000000
-2.810000 -2.545000 0.000000
-2.650000 -2.650000 0.000000
-2.545000 -2.810000 0.000000
-2.500000 -3.000000 0.000000
-2.545000 -3.190000 0.000000
-2.650000 -3.350000 0.000000
-2.810000 -3.455000 0.000000
18
16 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0xff0
```



```
16 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 0xff0
4 0 16 17 1 0xff0
4 1 17 18 2 0xff0
4 2 18 19 3 0xff0
4 3 19 20 4 0xff0
4 4 20 21 5 0xff0
4 5 21 22 6 0xff0
4 6 22 23 7 0xff0
4 7 23 24 8 0xff0
4 8 24 25 9 0xff0
4 9 25 26 10 0xff0
4 10 26 27 11 0xff0
4 11 27 28 12 0xff0
4 12 28 29 13 0xff0
4 13 29 30 14 0xff0
4 14 30 31 15 0xff0
4 15 31 16 0 0xff0
Cut-pyramid
8
0.250000 -5.427271 0.500000
0.250000 -5.927271 0.500000
-0.250000 -5.927271 0.500000
-0.250000 -5.427271 0.500000
0.500000 -5.177271 0.000000
0.500000 -6.177271 0.000000
-0.500000 -6.177271 0.000000
-0.500000 -5.177271 0.000000
6
4 0 1 2 3 0xf00
4 7 6 5 4 0xf00
4 0 4 5 1 0xf00
4 1 5 6 2 0xf00
4 2 6 7 3 0xf00
4 3 7 4 0 0xf00
door
4
0.636038 6.980460 0.843529
```

```
0.636038 6.980460 0.056808
-0.636038 6.980460 0.056808
-0.636038 6.980460 0.843529
2
4 0 1 2 3 0x001
4 3 2 1 0 0x001
doorway
6
-0.751193 7.120286 0.848529
0.778841 7.120292 0.848529
0.013824 7.120290 1.178401
-0.751193 6.555811 0.848529
0.778841 6.555817 0.848529
0.013824 6.555815 1.178401
5
3 0 1 2 0xff0
3 5 4 3 0xff0
4 0 3 4 1 0xff0
4 1 4 5 2 0xff0
4 2 5 3 0 0xff0
doorway
8
-0.754731 6.581469 0.842968
-0.754731 7.094628 0.842968
-0.619600 7.094628 0.842968
-0.619600 6.581469 0.842968
-0.754731 6.581468 0.049055
-0.754731 7.094627 0.049055
-0.619600 7.094627 0.049055
-0.619600 6.581468 0.049055
6
4 0 1 2 3 0xff0
4 7 6 5 4 0xff0
4 0 4 5 1 0xff0
4 1 5 6 2 0xff0
4 2 6 7 3 0xff0
4 3 7 4 0 0xff0
```



```
doorway
8
0.647248 6.581475 0.842968
0.647248 7.094634 0.842968
0.782379 7.094634 0.842968
0.782379 6.581475 0.842968
0.647248 6.581474 0.049055
0.647248 7.094633 0.049055
0.782379 7.094633 0.049055
0.782379 6.581474 0.049055
6
4 0 1 2 3 0xff0
4 7 6 5 4 0xff0
4 0 4 5 1 0xff0
4 1 5 6 2 0xff0
4 2 6 7 3 0xff0
4 3 7 4 0 0xff0
Pyramid
5
0.500000 -3.500000 0.014977
-0.500000 -3.500000 0.014977
-0.500000 -2.500000 0.014977
0.500000 -2.500000 0.014977
0.000000 -3.000000 2.125046
5
4 3 2 1 0 0xf0f
3 3 4 2 0xf0f
3 2 4 1 0xf0f
3 1 4 0 0xf0f
3 0 4 3 0xf0f
```

F

VImCvt (Image Format Converter)

Overview

VImCvt converts a still image file between two different image formats (including compressed images) with optional cropping, scaling, or conversion between color spaces.

VImCvt *[Options] InFile InFormat OutFormat [OutFile]*

Options	Described below
InFile	Name of input image
InFormat	Format of input image
OutFormat	Format of output image
OutFile	Optional name of output image

VImCvt searches for input files using the DOS environmental variable VIM. If the optional argument OutFile is not specified, the image will be saved in the current directory under the same name. OutFile can be specified to save the image under a different name or in a different directory.

Image names can be given with no extension, since the extension is implied by the format. If any extensions are supplied, they will be ignored.

The supported formats and their command line argument names are as follows:

Format	Description	File Extensions
9	DVI 9-bit format	.imx, .imy, .imz
8	DVI 8-bit CLUT format	.i8
16	DVI 16-bit format	.i16
24	DVI 24-bit format	.imx, .imy, .imz
c9	DVI 9-bit compressed	.cmx, .cmy, .cmz
c16	DVI 16-bit compressed	.c16
avs	DVI 9-bit AVSS format	.avs
pix	Lumena format	.pix
t16	Targa 16-bit uncompressed	.t16, .tga, .win
t32	Targa 32-bit uncompressed	.t32, .tga, .win
raw	Raw 24-bit data format	.x, .y, .z

In this table, X, Y, and Z represent one of the three triples RGB, YIQ, or YVU, which are the three color spaces supported by VImCvt. See the description of the option -i, which follows, for more information on the use of color spaces.

Raw format is a generic 24-bit format consisting of three data files (no headers), each containing 8-bit pixel component values for consecutive pixels (left to right, and top to bottom) in the image.

The input and output images must be 768x480 or smaller in resolution, or 512x480 or smaller if in Lumena or Targa format. On input, this restriction applies to the size before cropping.

The following command line options are available:

-cXLen,Ylen[,X,Y] Crop input image

Extracts a sub-image of size XLen by Ylen (at optional origin X,Y) from the input image. If X,Y are not supplied, they default to 0,0. If the input image format is 9, XLen, YLen, X, and Y should be multiples of 4.

-sXLen,Ylen Size of input image

Sets the size of the output image to XLen,YLen. The input image (after cropping, if the option -c is also specified) will be scaled to this resolution before conversion and output. XLen, YLen must be 768 x 480 or smaller. If the output image format is 9, XLen and YLen should be multiples of 4.

-rXLen,YLen Raw format input size

Specifies the input image size for raw format input. Since raw format image files have no headers containing image size information, option -r must be used to indicate the size of the image.

-iXYZ Input color space

-oXYZ Output color space

Specifies the color space of the input or output images. XYZ must be either RGB, YIQ, or YVU.

If a color space option is not present, VImCvt will use the following default color spaces:

RGB for Targa or Lumena format.

YVU for all other formats

Example

vimcvt -s128,120 puzzle.tga t16 16

This would convert a Targa 16 bit uncompressed (not RLE) image to a 128 x 120 pixel DVI 16-bit image for use with WorldToolKit. It will create a file called 'puzzle.i16'.

Index